Messaging for Many Applications

ZeroMQ

云时代极速消息通信库



[美] *Pieter Hintjens* 著

卢涛 李颖 译

O'REILLY®



電子工業出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

http://www.phei.com.cn

VISIT…

LANZAROTE
Caliente.COM

提供各种书籍pdf下载，如有需要，请联系 QQ: 461573687

PDF制作说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ: 461573687, 或者 QQ: 2404062482。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

备用QQ:2404062482

ZeroMQ
云时代极速消息通信库

ZeroMQ: Messaging for Many Applications

[美] Pieter Hintjens 著
卢涛 李颖 译

電子工業出版社
Publishing House of Electronics Industry

内 容 简 介

本书介绍ZeroMQ的API、套接字和模式的使用。通过建立应用程序来讲解如何使用ZeroMQ编程技术构建多线程应用程序，并创建自己的消息传递架构。本书设计了大量工作实例来实现请求-应答模式的高级使用和容错性，并对发布-订阅模式的性能、可靠性、状态分发与监控进行了扩展。

本书面向的读者是希望制作大规模分布式软件的专业程序员和有志于这方面研究的专业人士，旨在帮助他们解决大规模、可扩展、低成本、高效率的问题，书中还展现了ZeroMQ所需的网络和分布式计算概念。

©2013 by O'Reilly Media, Inc.
Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2015. Authorized translation of the English edition, 2013 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由O'Reilly Media, Inc.授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号　图字：01-2014-5267

图书在版编目（CIP）数据

ZeroMQ：云时代极速消息通信库 /（美）亨特金斯（Hintjens,P.）著；卢涛，李颖译．—北京：电子工业出版社，2015.3
书名原文：ZeroMQ: messaging for many applications
ISBN 978-7-121-25311-9

Ⅰ．①Z…　Ⅱ．①亨…　②卢…　③李…　Ⅲ．①计算机网络－软件工具　Ⅳ．①TP393.07

中国版本图书馆CIP数据核字（2014）第308036号

策划编辑：张春雨
责任编辑：刘　舫
封面设计：Randy Comer　张　健
印　　刷：北京丰源印刷厂
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787×980　1/16　印张：33.5　字数：793千字
版　　次：2015年3月第1版
印　　次：2015年3月第1次印刷
定　　价：108.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

To Noémie, Freeman, and Gregor.

译者序

ZeroMQ 是 iMatix 开发的以消息为导向的开源中间件库，它类似于标准 Berkeley 套接字，支持多种通信模式（扇出、发布 - 订阅、任务分配和请求 - 应答）和传输协议（进程内、进程间、TCP 和多播），可以用作一个并发框架。其核心由 C 编写，支持 C++、Java 等多种语言的 API，能在大部分现代平台上运行。它的运行速度很快，其异步模型具有高可扩展性。ZeroMQ 中的 Zero（零）代表简约并涵盖不同目标:零代理、零延迟、零管理、零成本等。

本书源自 ZeroMQ 的参考手册，通过社区共同开发而成，数百人为本书做出了贡献，包括用各种编程语言编写的示例。本书也为运作一个成功的社区提供了范例。

值得一提的是，本书作者 Pieter Hintjens 是 iMatix 公司的首席执行官，他于 2010 年被查出患了晚期癌症，动了多次手术并化疗 6 个月，但 2011 年仍然坚持继续完成这本书的写作。这种奋不顾身的专业精神值得我们学习。

感谢电子工业出版社计算机出版分社的张春雨编辑选择我们翻译本书，感谢刘舫编辑，她从专业的角度对译文进行了把关，并进行了许多润色，使之更具可读性。

感谢李绿霞、卢林、陈克非、李洪秋、张慧珍、李又及、卢晓瑶、陈克翠、汤有四、李阳、刘雯、贾书民、苏旭晖对本书翻译工作做出的贡献。

还要感谢我们的儿子卢〇一小朋友，他知道我们在翻译书稿就常常自己安静地读书和玩耍，还放弃了很多出去玩的机会，让我们能够专注于本书的翻译，本书的出版也有他的一份贡献。

最后希望这本书对读者有帮助。但由于译者经验和水平有限，译文中难免有不妥之处，恳请读者批评指正！

卢涛　李颖

2014 年 12 月 8 日

前言

ØMQ 的一百字概括

ØMQ（也称为 ZeroMQ、0MQ 或 zmq）看起来像一个可嵌入的网络库，但其作用就像一个并发框架。它为你提供了在各种传输工具，如进程内、进程间、TCP 和组播中进行原子消息传送的套接字。你可以使用各种模式实现 N 对 N 的套接字连接，这些模式包括扇出、发布 - 订阅、任务分配和请求 - 应答。它的速度足够快，因此可充当集群产品的结构。它的异步 I/O 模型提供了可扩展的多核应用程序，用异步消息来处理任务。它有多种编程语言的 API，并可运行在大多数操作系统上。ØMQ 是 iMatix（*http://www.imatix.com*）开发的，并在 LGPLv3 许可下开源。

零之禅

ØMQ 中的 Ø 关乎权衡。一方面，ØMQ 这个奇怪的名字使其在谷歌和 Twitter 上降低了知名度。另一方面，它惹恼了一些丹麦人（译者注：Ø 是丹麦语字母表中的字母），他们写给我们如下的东西，比如“ØMG røtfl”（译者注：把 Ø 替换成 O 后表示“天哪，笑翻了”）、“Ø 不是一个有趣的零！”（译者注：早期计算机输出中用这个符号表示数字 0，以便与字母 O 区分）和“*Rødgrød med Fløde!*”（这显然是一种侮辱，意思是“愿你的邻居是格伦德尔的直系后裔！”）（译者注：在古英国史诗《贝奥武夫》中，格伦德尔是一只雄性怪兽，这里作者好像是在误导，rødgrød med fløde，意为“浇了奶油的红莓布丁”，是个非常经典的丹麦语绕口令，它难倒了很多外国人，因为这短短的一句话中包括了三个“ø”（两种不同的发音方式）、咽喉擦音 r、重音 gr 组合和软化的 d（发音类似于英语 with 中的 th）。）这毁誉参半的两方面似乎是一个公平的交易。

最初，ØMQ 中的“零”是为了表示“零代理”和（尽可能接近）“零延迟”。但从那时起，

它已经涵盖了不同的目标：零管理、零成本、零浪费。更一般的，“零”是指贯穿项目的简约文化。我们通过消除复杂性，而不是通过公开新功能来增加威力。

本书的历程

在 2010 年的夏天，ØMQ 仍是一个名不见经传的小众库，它由相当简洁的参考手册和一个活跃但精炼的 wiki 来描述。那时，Martin Sustrik 和我坐在布拉迪斯拉发 Kyjev 酒店的酒吧里，密谋如何让 ØMQ 获得更广泛的普及。Martin 写了大部分的 ØMQ 代码，而我提供了资金并组织了社区。因为某种原因，我们一致认为 ØMQ 需要一个更简单的新网站，并需要为新用户写一本基本指南。

Martin 收集了一些要讲解的主题。由于在此之前，我连一行 ØMQ 代码都没写过，因此这本书的写作过程成了一个活学活用的见证。当我通过简单的例子进入复杂的工作时，我试着回答了很多在邮件列表上见过的问题。因为 30 年来我一直在构建大型架构，所以我有很多问题渴望用 ØMQ 来解决。令人惊讶的是，用 ØMQ 来解决问题所获得的结果大多是简单而优雅的，甚至当我用 C 语言工作时，学习 ØMQ 和用它解决实际问题也会使我感到一种纯粹的快乐，这使我在几年的停顿后，再一次回归编程领域。而往往，我们起初也不知道它“应该”如何完成某项工作，在使用过程中，我们不断地对 ØMQ 进行着改进。

因为从一开始，我就想把这本指南做成一个社区项目，所以我把它放到 GitHub 上，让别人通过提出请求的方式为本书做出贡献。这种做法被某些人认为是激进的，甚至是低俗的。我们实现了分工：我负责写作和制作原始的 C 语言示例，其他人帮助修复文本中的错误并把示例翻译成其他语言。

这种模式工作得比我想象的更好。现在你可以找到所有示例的多种语言版本，许多例子甚至有十几种语言版本。它是一块编程语言的罗塞塔石碑（译者注：刻有古埃及国王托勒密五世登基的诏书的石碑。上面用希腊文字、古埃及文字和当时的通俗体文字刻了同样的内容），这本身就是一个有价值的结果。我们建立了高分标准：翻译程度达到 80%，每种语言都有自己的指南。PHP、Python、Lua 和 Haxe 都达到了这个目标。人们要求我们创建 PDF 文件，我们就创建好后提供给他们。人们要求提供电子书，他们也如愿以偿了。目前，这本指南得到了大约一百多人的帮助。

该指南实现了普及 ØMQ 的目标。这种风格取悦了大多数人但也惹恼了一些人，这是自然的。2010 年 12 月，我在 ØMQ 和指南上的工作停顿了下来，因为我查出自己患了晚期癌症，动了多次手术并化疗 6 个月。当我在 2011 年年中再度拾起这项工作时，它开始在可以想象的最大的用例之一上爆发，即：在全球最大的电子公司的手机和平板电脑

上使用 ØMQ。

但是，本指南的目标从一开始就是一本印刷书籍。所以，在 2012 年 1 月我收到 Bill Lubanovic 发来的一封电子邮件，这是令人兴奋的，他把我介绍给 O'Reilly 公司的 Andy Oram 编辑，建议出一本 ØMQ 的图书。"当然！"我说，"可我在哪里签名呢？我需要付多少钱？呵呵，我能得到钱吗？所有我需要做的就是完成它吗？"

当然，只要 O'Reilly 发布一本 ØMQ 图书，其他出版社就会开始给潜在的作者发送电子邮件约稿。明年你可能会看到大量的 ØMQ 图书现身。这是很好的。我们的小众库已经成为主流，并值得拥有六英寸的书架空间。我要向其他 ØMQ 图书的作者致歉，我们已经设置了可怕的高门槛，我的建议是让你的书具有互补性，也许可以专注于一种特定的语言、平台或模式。

这就是社区的神奇和力量：如果你创建的社区能成为在某一个领域中的第一个社区，并保持健康发展，你就将永远占据这个领域的领导地位。

读者对象

本书是为专业程序员编写的，旨在帮助他们了解如何制作将主宰未来计算的大规模分布式软件。我们假设你能够阅读 C 代码，因为这里的大多数示例都用 C 语言编写（尽管 ØMQ 被用在许多语言中）。我们假设你关注规模，因为 ØMQ 解决了上述所有其他的问题。我们假设你需要用最少的成本获得最好的结果，因为否则你不会欣赏 ØMQ 所做的权衡。除了这些基本的背景，我们还试图为你呈现使用 ØMQ 将需要的所有网络和分布式计算的概念。

本书所使用的约定

本书使用以下印刷约定：

斜体（*Italic*）

用于表示新的术语、命令或命令行选项、URL、电子邮件地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序清单，以及段落中引用的程序元素，如变量、函数名、数据类型和环境变量。

等宽粗体（**`Constant width bold`**）

用于表示由用户输入的命令行。

等宽斜体（*Constant width italic*）

用于表示应被替换为用户提供的有意义的值的预留位置。



此图标表示提示、建议或一般的注意。

中文版书中切口以“▭>”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引中所列的页码为原英文版页码。

使用代码示例

代码示例都在在线存储库中，它位于 *https://github.com/imatix/zguide/tree/master/examples/*。你会发现每一个例子都被翻译成其他数种语言。这些例子都在 MIT/X11 下授权，详情请查看该目录中的 *LICENSE* 许可文件。本书正文解释了在各种情况下如何运行每个示例。

我们欢迎在使用代码示例时署名，但不强制要求这么做。一个署名通常包括标题、作者、出版商和 ISBN。例如：“*ZeroMQ* by Pieter Hintjens (O'Reilly). Copyright 2013 Pieter Hintjens, 978-1-449-33406-2.”。

如果你发现自己对书中代码的使用有失公允，或是违反了前述条款，敬请通过 *permissions@oreilly.com* 与我们联系。

Safari® Books Online



Safari Books Online 是一家按需所取的数字图书馆，它同时提供来自世界各地领先的技术和业务作者的书籍和视频两种形式的专业内容。

专业技术人员、软件开发人员、网页设计师，以及商业和创意专业人士使用 Safari 联机丛书作为研究，是解决问题、学习和认证培训的主要资源。

Safari Books Online 提供了一系列的产品组合和针对组织、政府机构和个人的定价方案。用户有机会在一个完全可搜索的数据库中访问成千上万的书籍、培训视频和即将出版的原稿，它们来自 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、

Course Technology 以及大量其他出版商。有关 Safari Books Online 的更多详细信息，请访问我们的网站。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

http:/bit.ly/ZeroMQ-OReilly

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站 *http://www.oreilly.com*。

在 Facebook 上找到我们：*http://facebook.com/oreilly*
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*
在 YouTube 上观看我们：*http://www.youtube.com/oreillymedia*

致谢

感谢 Andy Oram 使本书能够在 O'Reilly 出版并对此书进行了编辑。

感谢 Bill Desmarais、Brian Dorsey、Daniel Lin、Eric Desgranges、Gonzalo Diethelm、Guido Goldstein、Hunter Ford、Kamil Shakirov、Martin Sustrik、Mike Castleman、Naveen Chawla、Nicola Peduzzi、Oliver Smith、Olivier Chamoux、Peter Alexander、

Pierre Rouleau、Randy Dryburgh、John Unwin、Alex Thomas、Mihail Minkov、Jeremy Avnet、Michael Compton、Kamil Kisiel、Mark Kharitonov、Guillaume Aubert、Ian Barber、Mike Sheridan、Faruk Akgul、Oleg Sidorov、Lev Givon、Allister MacLeod、Alexander D' Archangel、Andreas Hoelzlwimmer、Han Holl、Robert G. Jakabosky、Felipe Cruz、Marcus McCurdy、Mikhail Kulemin、Dr. Gergö Érdi、Pavel Zhukov、Alexander Else、Giovanni Ruggiero、Rick "Technoweenie"、Daniel Lundin、Dave Hoover、Simon Jefford、Benjamin Peterson、Justin Case、Devon Weller、Richard Smith、Alexander Morland、Wadim Grasza、Michael Jakl、Uwe Dauernheim、Sebastian Nowicki、Simone Deponti、Aaron Raddon、Dan Colish、Markus Schirp、Benoit Larroque、Jonathan Palardy、Isaiah Peng、Arkadiusz Orzechowski、Umut Aydin、Matthew Horsfall、Jeremy W. Sherman、Eric Pugh、Tyler Sellon、John E. Vincent、Pavel Mitin、Min RK、Igor Wiedler、Olof Åkesson、Patrick Lucas、Heow Goodman、Senthil Palanisami、John Gallagher、Tomas Roos、Stephen McQuay、Erik Allik、Arnaud Cogoluègnes、Rob Gagnon、Dan Williams、Edward Smith、James Tucker、Kristian Kristensen、Vadim Shalts、Martin Trojer、Tom van Leeuwen、Hiten Pandya、Harm Aarts、Marc Harter、Iskren Ivov Chernev、Jay Han、Sonia Hamilton、Nathan Stocks、Naveen Palli 和 Zed Shaw 对这项工作做出的贡献。

感谢 Martin Sustrik 多年来在 ZeroMQ 上所做的令人难以置信的工作。

感谢 Stathis Sideris 为 Ditaa（*http://www.ditaa.org*）做出的贡献。

关于作者

30 年前，Pieter Hintjens 从制作视频游戏起家，并持续构建各种软件产品至今。他信奉“软件的本质就是人的本质”的原则，他现在专注于通过“社会结构”建设社区、写作和帮助他人使用 ZeroMQ 获利。

他曾在一个与软件专利做斗争的大型非政府组织——FFII 中担任了两年的会长。他曾是 Wikidot 的 CEO，欧洲专利大会的创始人以及数字标准组织的创始人。Pieter 会说英语、法语、荷兰语，并对其他十多种语言略知一二。Pieter 与他美丽的妻子和三个可爱的孩子生活在比利时布鲁塞尔，他与当地的西非鼓乐队一起演奏，并喜欢去各地旅行。

封面介绍

本书封面的动物是 fourhorn sculpin 鱼（Myoxocephalus quadricornis）。这种鱼主要生活在环绕北美和欧亚大陆北部的北极近海，也可以在欧洲的一些淡水湖泊中找到它们。这种鱼以它的头部的 4 个骨性突起而命名。

fourhorn sculpin 鱼的鱼身暗黑而稍扁平，眼睛接近其头顶部，它的骨盆硕大，并且有独特的大嘴。它的体长通常可达约 30 厘米。一种区别这个物种的雄性和雌性的方法是，雄性的肚子为黄褐色而雌性的肚子为白色。这种鱼主要是以海底的生物体为食，如甲壳类动物和鱼卵。

fourhorn sculpin 鱼在冬季繁殖。在此期间，雄鱼通常会挖一个坑，雌鱼会把所有的卵放进去。一旦雌鱼在坑里完成产卵，雄鱼就会在三个月的潜伏期中守护卵。

封面图片取自约翰逊的 *Natural History* 一书。

目录

前言 xix

第 1 部分 学习如何使用 ØMQ 来开展工作

第 1 章 基础知识 3

修复这个世界 3
本书的读者对象 5
获取示例 5
问过就必有收获 5
在字符串上的小注解 10
版本报告 12
获得消息 12
分而治之 16
用 ØMQ 编程 21
  获取正确的上下文 22
  执行彻底的退出 22
为什么我们需要 ØMQ 23
套接字的可扩展性 27
从 ØMQ v2.2 升级到 ØMQ v3.2 28
警告：不稳定的典范！ 29

第 2 章 套接字和模式 31

套接字 API 32
把套接字接入网络拓扑 32
使用套接字来传输数据 34
单播传输 35
ØMQ 不是一个中性载体 35
I/O 线程 36
消息传递模式 37
高级别消息传递模式 38
处理消息 38
处理多个套接字 41
多部分消息 44
中间层及代理 45
动态发现问题 46
共享队列（DEALER 和 ROUTER 套接字） 48
ØMQ 的内置代理功能 53
传输桥接 55
处理错误和 ETERM 56
处理中断信号 61
检测内存泄漏 63
使用 ØMQ 编写多线程程序 64
线程间信令（PAIR 套接字） 69
节点协调 71
零拷贝 75
发布 - 订阅消息封包 76
高水位标记 78
消息丢失问题的解决方案 80

第 3 章 高级请求 - 应答模式 83

请求 - 应答机制 83
简单的应答封包 84
扩展的应答封包 84
这有什么好处呢 87

请求 - 应答套接字回顾 88
请求 - 应答组合 88
REQ 到 REP 组合 89
DEALER 到 REP 组合 89
REQ 到 ROUTER 组合 90
DEALER 到 ROUTER 组合 90
DEALER 到 DEALER 组合 90
ROUTER 到 ROUTER 组合 90
无效组合 91
探索 ROUTER 套接字 91
身份和地址 92
ROUTER 错误处理 93
负载均衡模式 94
ROUTER 代理和 REQ 工人 95
ROUTER 代理及 DEALER 工人 97
负载均衡的消息代理 98
用于 ØMQ 的一个高级别的 API 105
高级别 API 的特点 107
CZMQ 高级别 API 108
异步客户端 / 服务器模式 115
能够工作的示例：跨代理路由 120
建立详情 120
单集群架构 121
扩展到多个集群 122
联盟与对等比较 124
命名规范 126
状态流原型 127
本地流和云端流原型 130
总结 137
第 4 章 可靠的请求 - 应答模式 147
什么是“可靠性” 147
可靠性设计 148

客户端可靠性（懒惰海盗模式）......149
基本可靠队列（简单海盗模式）......154
健壮的可靠队列（偏执海盗模式）......157
信号检测......166
置若罔闻地将它关闭......166
单向信号检测......167
乒乓信号检测......167
针对偏执海盗的信号检测......168
合同和协议......170
面向服务的可靠队列（管家模式）......170
异步管家模式......195
服务发现......201
幂等服务......203
断开连接的可靠性（泰坦尼克模式）......203
高可用性对（双星模式）......216
详细需求......218
避免脑裂症状......220
双星实现......221
双星反应器......228
无代理可靠性（自由职业者模式）......234
模型一：简单的重试和故障转移......235
模型二：粗暴猎枪屠杀......238
模式三：复杂和讨厌的......244
结论......256
第 5 章 高级发布 - 订阅模式......257
发布 - 订阅模式的优点和缺点......257
发布 - 订阅跟踪（特浓咖啡模式）......259
最后一个值缓存......262
慢速订阅者检测（自杀蜗牛模式）......267
高速订阅者（黑盒模式）......270
可靠的发布 - 订阅（克隆模式）......272
集中式与分散式......273

将状态表示为键 - 值对 273
得到带外的快照 284
重新发布来自客户端的更新 290
处理子树 295
临时值 298
使用反应器 306
在双星模式中添加可靠性 311
集群的散列映射协议 321
构建一个多线程栈和 API 325

第 2 部分 使用 ØMQ 的软件工程

第 6 章 ØMQ 社区 341

ØMQ 社区的架构 342
如何制作真正的大型架构 343
软件架构的心理学 344
合同 346
过程 348
疯狂，美丽，并且容易 348
陌生人，遇见陌生人 349
无限的财富 349
照管和培育 350
ØMQ 过程：C4 351
语言 351
目标 352
热身 354
许可和所有权 355
对补丁程序的要求 356
开发过程 357
建立稳定的版本 361
公共合同的演变 362
一个实际例子 364
Git 分支是有害的 368

简单性与复杂性的对比......369
更改延迟......369
学习曲线......369
出故障的成本......369
前期协调......369
可扩展性......370
惊奇和期望......370
参与的经济学......370
在冲突中的强壮性......370
隔离的保证......370
能见度......371
结论......371
为创新而设计......371
双桥传说......371
ØMQ 的路线图是如何失去的......372
垃圾桶化的设计......374
复杂化的设计......376
简约化的设计......377
职业倦怠......379
成功模式......380
懒惰的完美主义者......381
仁慈暴君......381
天和地......381
门户开放......381
大笑的小丑......382
留心的将军......382
社会工程师......382
不朽的园丁......382
滚石......382
海盗帮......383
快闪族......383
加那利看守......383
执行绞刑的刽子手......383

历史学家 383
煽动者 384
神秘人 384

第 7 章 使用 ØMQ 的高级架构 385

用于弹性设计的面向消息模式 386
第 1 步：内部化的语义 387
第 2 步：描绘一个粗略的架构 387
第 3 步：决定合同 388
第 4 步：编写一个最小的端到端解决方案 388
第 5 步：解决一个问题，然后重复 389
Unprotocol 389
合同是艰难的 390
如何编写 Unprotocol 391
为什么使用 GPLv3 的公开规范 392
使用 ABNF 393
廉价或讨厌的模式 393
序列化数据 395
ØMQ 组帧 396
序列化语言 396
序列化库 397
手写的二进制序列化 399
代码生成 400
传输文件 406
状态机 417
使用 SASL 认证 424
大型文件发布：FileMQ 426
为什么要制作 FileMQ 426
最初的设计切片：API 426
最初的设计切片：协议 427
构建和尝试 FileMQ 429
内部架构 430
公共 API 431

设计说明 432
配置 433
文件稳定性 434
递交通知 434
符号链接 435
恢复和后期加入者 435
测试用例：曲目工具 437
得到一个官方端口号 439

第 8 章 分布式计算的框架 441

用于现实世界的设计 442
无线网络的秘密生活 443
为什么网状网络现在还没出现 444
一些物理知识 445
现状是什么 446
结论 448
发现 448
通过原始套接字先发制人的发现 448
使用 UDP 广播协同发现 450
一台设备上的多个节点 455
设计 API 456
关于 UDP 的更多内容 465
分拆一个库项目 466
点对点消息传递 467
UDP 信标帧 467
真正的对等连接（和谐模式） 469
检测失踪 471
群发消息 472
测试与模拟 474
使用断言 474
前期测试 475
Zyre 测试仪 476
测试结果 479

跟踪活动......481
处理阻塞节点......481
分布式日志记录和监视......484
一个合理的最小实现......485
协议断言......488
二进制日志记录协议......489
内容分发......490
编写 Unprotocol......493
结论......494

第 9 章 后记......497

番外篇......497
Rob Gagnon 的故事......497
Tom van Leeuwen 的故事......497
Michael Jakl 的故事......498
Vadim Shalts 的故事......498
本书是如何诞生的......499
消除摩擦......500
许可......502

索引......503

第 1 部分 1

学习如何使用ØMQ来开展工作

在本书的第 1 部分，你将学习如何使用 ØMQ。我们将讨论它的基础知识、API、不同的套接字类型以及它们的工作原理、可靠性，和许多可以在你的应用程序中使用的模式。通过从头到尾阅读正文，并练习示例，你将得到最大的收获。

第 1 章 3

基础知识

修复这个世界

如何来解释 ØMQ 呢？我们中有些人从它的丰功伟绩开始讲述。它是激素上的套接字。它就像具有路由的邮箱。它很快！另一些人想分享他们的启蒙时刻——那一切都变得明显得“稀里哗啦”，一切都发生根本变化的顿悟时刻。事情变得简单了。复杂性消失了。它打开了思路。其他人则试图通过比较来解释。它更小巧，更简单，但看起来仍然很熟悉。就个人而言，我喜欢去回忆我们到底为什么创造 ØMQ，因为这也可能是读者们现在最想知道的。

编程是装扮成艺术的科学，因为我们大多数人并不了解软件的本质，并且即使有人了解它的本质，也很少传授给他人。软件的本质并不是算法、数据结构、语言和抽象。这些都只是我们制作、使用并扔掉的工具。软件的真正本质正是人的本质。

具体来说，它与我们在处理复杂问题时的局限性和我们共同努力分片解决大问题的愿望有关。这是程序设计的科学：制作人们容易理解和使用的构件，而人们将共同努力来解决非常大的问题。

我们生活在一个互联的世界，而现代软件必须指引这个世界。因此，解决未来非常大的问题的构件都是互联和大规模并行的。“强大而沉默”的代码不再是足够的。代码与代码之间必须交流。代码必须健谈、善于交流，并能够良好地互联。代码必须像人脑那样运行，通过亿万个神经元相互发射消息给对方，就像一个无中央控制，无单点故障，但能够解决非常难处理的问题的大规模并行网络。而且未来的代码看起来像人脑，这并不是偶然的，因为在一定程度上，每个网络的端点就是人脑。

如果你曾经做过线程、协议或网络上的任何工作，你就会意识到这几乎是不可能的。这 4

是一个梦想。当你开始处理现实生活中的情形时，即使使用几个套接字连接少量的程序都让人不胜其烦。连接亿万个程序，情况会怎么样呢？这个成本是不可想象的。连接电脑是如此困难，以至于创造做到这一点的软件和服务都成了价值数十亿美元的产业。

所以，现实的环境是，我们的布线能力比起使用它的能力要领先多年。在 20 世纪 80 年代，曾爆发过一次软件危机，当时，领先的软件工程师如弗雷德·布鲁克斯认为，没有“银弹”（*http://en.wikipedia.org/wiki/No_Silver_Bullet*）可以承诺将生产率、可靠性或简洁性提高哪怕只是一个数量级。

布鲁克斯没赶上自由和开放源码软件大发展的时代，这种软件解决了危机，使我们能够有效地共享知识。今天，我们面临着另一个软件危机，但对它我们谈论得不太多。只有最大、最富有的公司才有能力建立互联的应用程序。虽然有云，但它是专有的。我们的数据和知识都从我们的个人电脑流失到我们不能访问，并且无法抗衡的云服务器里了。谁拥有我们的社交网络？这就像将主机 -PC 的革命又颠倒了过来。

我们可以将政治哲学留给另一本书（*http://swsi.info*）（译者注：本书作者写的 *Culture & Empire Digital Revolution* 一书）。问题的关键是，虽然互联网提供了互联大量代码的潜力，但现实是，这对于我们大多数人来说是遥不可及的，而许多如此有重要意义的问题（在健康、教育、经济、交通等方面）仍然未解决，这都是因为代码没有办法互联，因此也没有办法集合大家的智慧来解决这些问题。

人们已经做了许多尝试来解决软件互联的挑战。有成千上万的 IETF 规范，各自解决谜题的各个部分。对于应用程序开发人员来说，HTTP 也许是一个已经足够简单，能够开展工作的解决方案，但它鼓励开发人员和架构师考虑大型服务器和瘦呆的客户端模式，这无疑使得问题更加恶化了。

所以，现在人们仍然使用原始的 UDP 和 TCP、专有协议、HTTP 和 WebSocket 来连接应用程序。这项工作仍然是痛苦、缓慢、难以扩展，而且基本上是集中的。分布式对等体系结构大多是用来玩的，而不是用在工作上的。试问有多少应用程序使用 Skype 或者 BT 来交换数据呢？

这把我们带回到编程的科学这一话题。为了修复这个世界，我们需要做两件事情。一是解决“如何将任何地方的任何代码连接起来”的一般问题。二是将它包装在人们容易理解和使用尽可能简单的构件中。

这听起来简单得可笑。也许本来就是，但那就是问题的所在。

本书的读者对象

我们假设你使用的是最新的 ØMQ 3.2 版本。我们假设你使用的是 Linux 机器或类似的电脑。我们假设你可以或多或少地阅读 C 代码，因为这是示例的默认语言。我们假设，当我们写诸如 PUSH 或 SUBSCRIBE 的常量时，如果编程语言需要它们，你能想到它们其实是 ZMQ_PUSH 或 ZMQ_SUBSCRIBE。

获取示例

本书的示例都存放在这本书的 Git 存储库（*https://github.com/imatix/zguide*）中。获取所有例子最简单的方法是克隆这个存储库：

```
git clone --depth=1 git://github.com/imatix/zguide.git
```

接着，浏览 *examples* 子目录，你会发现按语言分类的示例。如果你使用的语言有例子缺失，我们希望你提交一个翻译版本（*http://zguide.zeromq.org/main:translate*）。本书之所以变得如此有用，这要归功于很多人的工作。所有的例子都按照 MIT/X11 许可授权。

问过就必有收获

因此，让我们先从一些代码起步。当然，我们将以一个“Hello World”的例子开始。我们会制作一个客户端和服务器，客户端发送“Hello”到服务器，服务器用“World”来应答（参见图 1-1）。示例 1-1 给出了服务器的 C 代码，这将在 5555 端口上打开一个 ØMQ 套接字，读取请求，并用“World”应答每个请求。

示例1-1：Hello World服务器（hwserver.c）

```
//
//  Hello World 服务器
//  将 REP 套接字绑定到 tcp://*:5555
//  期望从客户端收到“Hello”，用“World”来应答
//
#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main (void)
{
    void *context = zmq_ctx_new ();

    //  与客户端交流的套接字
```

```
        void *responder = zmq_socket (context, ZMQ_REP);
        zmq_bind (responder, "tcp://*:5555");

6   while (1) {
        //  等待来自客户端的下一个请求
        zmq_msg_t request;
        zmq_msg_init (&request);
        zmq_msg_recv (&request, responder, 0);
        printf ("Received Hello\n");
        zmq_msg_close (&request);

        //  做某项“工作”
        sleep (1);

        //  将应答发回给客户端
        zmq_msg_t reply;
        zmq_msg_init_size (&reply, 5);
        memcpy (zmq_msg_data (&reply), "World", 5);
        zmq_msg_send (&reply, responder, 0);
        zmq_msg_close (&reply);
    }
    //  我们永远不会到达这里，不过如果到了这里，这就是我们结束它的方法
    zmq_close (responder);
    zmq_ctx_destroy (context);
    return 0;
}
```

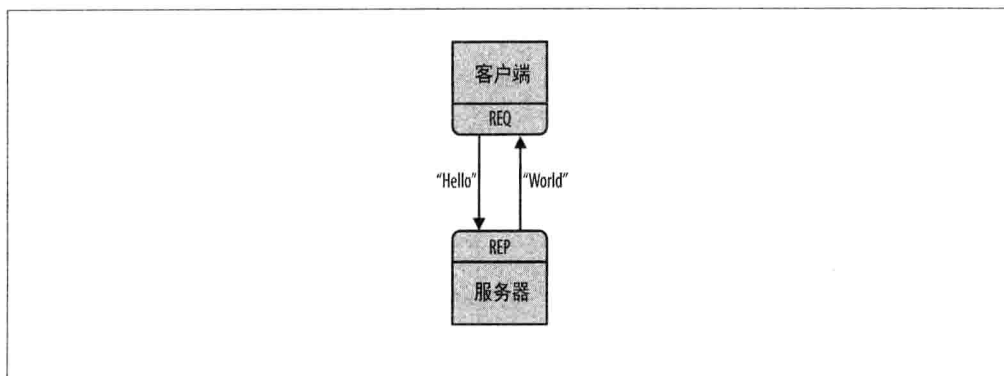


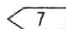
图1-1：请求-应答

REQ-REP 套接字对是步调一致的。客户端在一个循环中（或一次，根据需要而定）先发出 zmq_msg_send()，然后再发出 zmq_msg_recv()。任何其他序列（例如，一行中发

送两个消息）将导致从 send 或 recv 的代码返回 -1。同样的，服务器先发出 zmq_msg_recv()，然后再发出 zmq_msg_send()，按照这个顺序，根据需要多次重复。

ØMQ 采用 C 作为它的参考语言，这也是我们的例子将使用的主要语言。如果你正在联机读这本书的电子版，例子下面的链接将引导你到其他编程语言的翻译版本。对于阅读印刷版的读者，示例 1-2 显示的是同一个服务器程序在 C++ 中的样子。

示例1-2：Hello World服务器（hwserver.cpp）




```
//
//  用 C++ 编写的 Hello World 服务器
//  将 REP 套接字绑定到 tcp://*:5555
//  期望从客户端收到“Hello”，用“World”来应答
//
#include <zmq.hpp>
#include <string>
#include <iostream>
#include <unistd.h>

int main () {
    //  准备上下文和套接字
    zmq::context_t context (1);
    zmq::socket_t socket (context, ZMQ_REP);
    socket.bind ("tcp://*:5555");

    while (true) {
        zmq::message_t request;

        //  等待来自客户端的下一个请求
        socket.recv (&request);
        std::cout <<"Received Hello" << std::endl;

        //  做某项“工作”
        sleep (1);

        //  将应答发回给客户端
        zmq::message_t reply (5);
        memcpy ((void *) reply.data (), "World", 5);
        socket.send (reply);
    }
    return 0;
}
```

你会发现在 C 和 C++ 中的 ØMQ API 很相似。在类似 PHP 的语言中，我们甚至可以隐藏更多东西，从而使代码的可读性更好，如示例 1-3 所示。

示例1-3：Hello World服务器（hwserver.php）

```
<?php
/*
 *  Hello World 服务器
 *  将 REP 套接字绑定到 tcp://*:5555
 *  期望从客户端收到 “Hello”，用 “World” 来应答
 * @author Ian Barber <ian(dot)barber(at)gmail(dot)com>
 */

$context = new ZMQContext(1);

//  与客户端交流的套接字
8 $responder = new ZMQSocket($context, ZMQ::SOCKET_REP);
$responder->bind("tcp://*:5555");

while (true) {
    //  等待来自客户端的下一个请求
    $request = $responder->recv();
    printf ("Received request: [%s]\n", $request);

    //  做某项 “工作”
    sleep (1);

    //  将应答发回给客户端
    $responder->send("World");
}
```

客户端代码如示例 1-4 所示。

示例1-4：Hello World客户端（hwclient.c）

```
//
//  Hello World 客户端
//  将 REQ 套接字连接到 tcp://localhost:5555
//  发送 “Hello” 给服务器，期待传回 “World”
//
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main (void)
{
    void *context = zmq_ctx_new ();

    //  与服务器交流的套接字
```

```
    printf ("Connecting to hello world server...\n");
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5555");

//      int request_nbr;
//      for (request_nbr = 0; request_nbr != 10; request_nbr++) {
//          zmq_msg_t request;
//          zmq_msg_init_size (&request, 5);
//          memcpy (zmq_msg_data (&request), "Hello", 5);
//          printf ("Sending Hello %d...\n", request_nbr);
//          zmq_msg_send (&request, requester, 0);
//          zmq_msg_close (&request);
//
//          zmq_msg_t reply;
//          zmq_msg_init (&reply);
//          zmq_msg_recv (&reply, requester, 0);
//          printf ("Received World %d\n", request_nbr);
//          zmq_msg_close (&reply);
//      }
    sleep (2);
    zmq_close (requester);
    zmq_ctx_destroy (context);
    return 0;
}
```


现在，这看起来简单得都不现实了，但 ØMQ 套接字就是你将一个平常的 TCP 套接字，注入从一个前苏联原子能秘密研究工程盗窃的放射性同位素的混合物，再用 1950 年的宇宙射线轰击它，并把它交到一个吸毒成瘾的、带有严重伪装物癖的、穿着氨纶的肌肉鼓鼓的漫画书的作者手中（参见图 1-2），你最终会得到的东西。是的，ØMQ 套接字是网络世界中拯救世界的超级英雄。（译者注：漫画书中的超级英雄经常是因为受到了射线辐射或者吃了特殊药物才具有了超能力。）

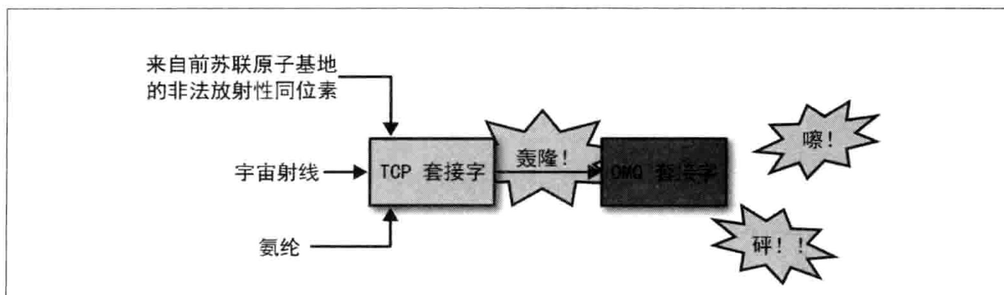


图1-2：一个可怕的事故……

你可以一下子向这台服务器抛出数以千计的客户端，而它会继续正常地快速工作。为了好玩，尝试先启动客户端，然后再启动服务器，看看这一切是如何仍然正常地工作的，然后花一秒钟思考这意味着什么。

让我们简单地解释一下这两个程序实际上在做什么。它们创建了用来工作的 ØMQ 上下文和一个套接字。不要担心不明白这些词的含义，以后你会知道的。服务器将 REP（应答）套接字绑定到 5555 端口，然后在一个循环中等待请求，并且每次用一个答复来响应。客户端发送一个请求，并读取从服务器返回的应答。

如果你终止服务器（按 Ctrl-C 组合键）并重新启动它，客户端将无法正常恢复。从崩溃的过程中恢复不太容易。因为做一个可靠的请求 - 应答流是很复杂的，所以我们在第 4 章之前都不会涉及它。

幕后有很多事情发生，但对程序员有意义的是，代码有多么简短而赏心悦目，以及即使是在负载很重时，它有多么不容易崩溃。这是请求 - 应答模式，可能是最简单的使用 ØMQ 的方法。它对应于 RPC（远程过程调用）和传统的客户端 / 服务器模型。

10

在字符串上的小注解

除字节大小外，ØMQ 对你发送的数据一无所知。这意味着你有责任安全地格式化数据，使得应用程序可以读取它。针对对象和复杂的数据类型做这种处理有专门的库，如协议缓冲区来完成。但即使是字符串，你也需要小心。

在 C 和其他一些语言中，字符串是用空字节来终止的。我们可以用额外的空字节来发送类似“HELLO”的字符串：

```
zmq_msg_init_data (&request, "Hello", 6, NULL, NULL);
```

但是，如果用另一种语言发送一个字符串，它有可能不会包含空字节。例如，当我们在 Python 中发送相同的字符串时，我们这样做：

```
socket.send ("Hello")
```

那么进入线路的是一个长度（对于短字符串，这个长度占用 1 字节）和作为单独字符的字符串内容（参见图 1-3）。

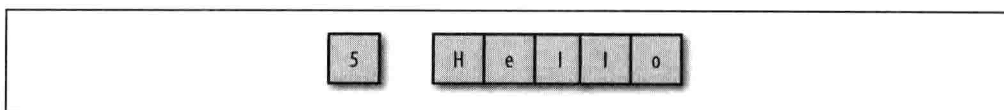


图1-3：一个ØMQ字符串

如果从一个 C 程序中读取这个字符串，会得到看起来像一个字符串，并且行为偶然像一个字符串的东西（如果碰巧 5 个字节后面跟着一个傻傻的潜伏空字符），但它不是正确的字符串。当你的客户端和服务器未就字符串格式达成一致时，就会得到奇怪的结果。

当从 ØMQ 用 C 接收字符串数据时，你根本无法相信它是安全地终止的。每一次你读到一个字符串时，都应该分配一个包含一个额外字节的新缓冲区，复制该字符串，并用正确的空字符来终止它。

因此，让我们建立“ØMQ 字符串是指定长度的，并且在线路上发送时不含结尾空字符”的规则。在最简单的情况下（我们将在例子中做到这一点），一个 ØMQ 字符串整齐地映射到一个 ØMQ 消息帧，它看起来就像图 1-3 所示——一个长度和一些字节。

下面是在 C 中接收 ØMQ 字符串，并将其作为一个有效的 C 字符串交付给应用程序时，我们需要做的工作：

```
//  从套接字接收 ØMQ 字符串并将其转换成 C 字符串
static char *
s_recv (void *socket) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    int size = zmq_msg_recv (&message, socket, 0);
    if (size == -1)
        return NULL;
    char *string = malloc (size + 1);
    memcpy (string, zmq_msg_data (&message), size);
    zmq_msg_close (&message);
    string [size] = 0;
    return (string);
}
```

这就编写出了一个非常方便的辅助函数。按照制作“可以有利可图地重用的东西”的精神，我们可以编写一个类似的、以正确的 ØMQ 格式发送字符串的 `s_send()` 函数，并将它打包到我们可以复用的头文件中。

其结果是 *zhelpers.h*，它可以让我们用 C 编写更赏心悦目和更小巧的 ØMQ 应用。它是一段相当长的源代码，并且只有 C 语言开发者对它有兴趣，所以请在休闲时间阅读它。（译者注：参见本书的示例代码网站 *https://github.com/imatix/zguide/blob/master/examples/C/zhelpers.h*。）

版本报告

ØMQ 确实有几种版本，而且经常有这种情况，如果你遇到一个问题，它很可能已在某个以后的版本中修复了。所以，确切地知道你实际链接的 ØMQ 版本是一个有用的技巧。示例 1-5 是一个很小的程序，但它可以让你做到这一点。

示例1-5：ØMQ版本报告（version.c）

```
//
//  报告 ØMQ 版本
//
#include "zhelpers.h"

int main (void)
{
    int major, minor, patch;
    zmq_version (&major, &minor, &patch);
    printf ("Current 0MQ version is %d.%d.%d\n", major, minor, patch);

    return EXIT_SUCCESS;
}
```

获得消息

第二个经典模式是单向的数据发布，其中一台服务器将更新推到一组客户端。让我们来看一个推送天气更新，包括邮政编码、温度和相对湿度的例子。我们将生成随机值，就像真正的气象站那样。

12 示例 1-6 显示了服务器的代码。我们将为这个应用程序使用 5556 端口。

示例1-6：天气更新服务器（wuserver.c）

```
//
//  天气更新服务器
//  将 PUB 套接字绑定到 tcp://*:5556
//  发布随机的天气更新
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和发布者
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    int rc = zmq_bind (publisher, "tcp://*:5556");
```

```
    assert (rc == 0);
    rc = zmq_bind (publisher, "ipc://weather.ipc");
    assert (rc == 0);

    //  初始化随机数发生器
    srandom ((unsigned) time (NULL));
    while (1) {
        //  获取会愚弄老板的值
        int zipcode, temperature, relhumidity;
        zipcode     = randof (100000);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;

        //  发送消息给所有订阅者
        char update [20];
        sprintf (update, "%05d %d %d", zipcode, temperature, relhumidity);
        s_send (publisher, update);
    }
    zmq_close (publisher);
    zmq_ctx_destroy (context);
    return 0;
}
```

这个更新流既没有起点也没有终点，它就像一个永无休止的广播（参见图 1-4）。

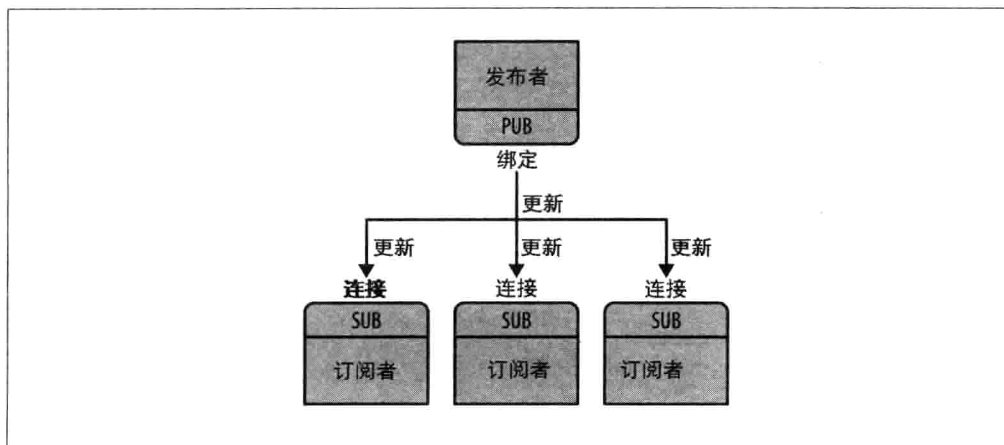


图1-4：发布-订阅

示例 1-7 显示了客户端应用程序，它监听更新的数据流，并抓取任何与特定邮政编码（默认情况下是纽约市，因为这是一个开始任何冒险的好地方）相关的信息。

13 示例1-7：天气更新客户端（wuclient.c）

```
//
//  天气更新客户端
//  将 SUB 套接字连接到 tcp://localhost:5556
//  收集天气更新并找出邮政编码所在地区的平均温度
//
#include "zhelpers.h"

int main (int argc, char *argv [])
{
    void *context = zmq_ctx_new ();

    //  与服务器交流的套接字
    printf ("Collecting updates from weather server...\n");
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    int rc = zmq_connect (subscriber, "tcp://localhost:5556");
    assert (rc == 0);

    //  订阅邮政编码，默认是 NYC, 10001
    char *filter = (argc > 1)? argv [1]: "10001";
    rc = zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, filter, strlen (filter));
    assert (rc == 0);

    //  处理 100 个更新
    int update_nbr;
    long total_temp = 0;
    for (update_nbr = 0; update_nbr < 100; update_nbr++) {
        char *string = s_recv (subscriber);

        int zipcode, temperature, relhumidity;
14      sscanf (string, "%d %d %d",
            &zipcode, &temperature, &relhumidity);
        total_temp += temperature;
        free (string);
    }
    printf ("Average temperature for zipcode '%s' was %dF\n",
        filter, (int) (total_temp / update_nbr));

    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

请注意，当你使用一个 SUB 套接字时必须使用 zmq_setsockopt() 和 SUBSCRIBE 设置一个订阅，如上面的代码所示。如果你没有设置任何订阅，就不会得到任何消息。这

是初学者常犯的一个错误。订阅者可以设置许多的订阅，它们被累加在一起。也就是说，如果某个更新匹配任何订阅，那么订阅者都会接收到它。订阅者也可以取消特定的订阅。订阅经常但不一定是一个可打印的字符串。要了解这是如何工作的，请参见 zmq_setsockopt()。

该 PUB-SUB 套接字对是异步的。客户端在一个循环中（或一次，根据需要而定）执行 zmq_msg_recv()。试图将消息发送到一个 SUB 套接字会导致错误。同样，服务根据需要多次执行 zmq_msg_send()，但不能在 PUB 套接字上执行 zmq_msg_recv()。

从理论上讲，使用 ØMQ 套接字不关心哪端连接和哪端绑定。然而，在实践中有未在文档中记录的差异，我稍后就会提到它。就目前而言，只要绑定 PUB 并连接 SUB 就可以了，除非你的网络设计使得这行不通。

关于 PUB-SUB 套接字还有一件更重要的事情需要了解：你不知道订阅者开始得到消息的精确时间。即使你启动一个订阅者，稍等片刻，然后再启动发布者，*订阅者也总是会错过发布者发送的第一个消息*。这是因为当订阅者连接到发布者时（这需要的时间很短，但非零），发布者可能已经将消息发送出去了。

这种“慢木匠”症状经常会击中足够多的人，那我们要详细解释一下。请记住，ØMQ 执行异步 I/O（即，在后台）。假设你有两个节点执行此操作，顺序如下：

- 订阅者连接到一个端点，并接收和清点消息。
- 发布者绑定到一个端点，并立即发送 1000 条消息。

订阅者很可能不会收到任何东西。你会眨眼，检查你是否设定了一个正确的过滤器，然后再试一次，而订阅者仍然没有收到任何东西。

建立 TCP 连接包含会花几毫秒（msec）的握手，这取决于你的网络和节点间的跳数。在这段时间里，ØMQ 可以发送很多消息。为了便于讨论，假定建立连接需要 5 毫秒，而且同一个链路每秒可以处理 1M 的消息。在订阅者连接到发布者的这 5 毫秒的时间内，发布者只需要花费 1 毫秒就能发送出那 1K 的消息。 15

在第 2 章中，我们将解释如何来同步发布者和订阅者，这样就不会启动数据发布，直到订阅者真正连接并准备就绪。有一个简单（愚蠢）的方式来延迟发布，这就是休眠（sleep）。但是，不要在实际应用程序中这么做，因为它非常脆弱，并且是不雅和缓慢的。利用休眠自己来检验发生了什么，然后阅读第 2 章，看看如何正确地做到这一点。

同步的替代办法是简单地假定发布的数据流是无限的，它没有起点也没有终点。人们还假设订阅者不关心它启动前发生了什么事情。这就是我们建立天气客户端例子的方式。

因此，客户端订阅它选择的邮政编码并收集针对该邮政编码的 1000 个更新。如果邮政编码是随机分布的，这意味着大约有 1000 万个更新从服务器发出。你可以先启动客户端，然后再启动服务器，而客户端将继续工作。你可以停止并重新启动服务器任意次，而客户端将继续工作。当客户端收集了 1000 个更新后，它就会计算并输出其平均值，然后退出。

关于发布 - 订阅模式的几个要点：

- 一个订阅者可以连接到多个发布者，每次使用一个 connect 调用。那么数据将交错到达（“公平排队”），因此，没有任何一个发布者能淹没其他发布者。
- 如果一个发布者没有连接的订阅者，那么它会简单地丢弃所有消息。
- 如果你使用的是 TCP 并且订阅者是慢速的，那么消息将在发布方排队。我们将在下一章了解如何通过使用“高水位线”来针对这种情况保护发布者。
- 从 ØMQ v3.x 开始，在使用连接的协议（tcp 或 ipc）时，过滤发生在发布方。使用 epgm 协议，过滤发生在订阅方。但在 ØMQ v2.x 版本中，所有过滤都发生在订阅方。

下面列出我的笔记本电脑接收和过滤 10M 消息所用的时间，这是一台 2011 年的英特尔 I7——快，但没什么特别的：

```
ph@nb201103:~/work/git/zguide/examples/c$ time wuclient
Collecting updates from weather server...
Average temperature for zipcode '10001' was 28F
16 real    0m4.470s
user    0m0.000s
sys     0m0.008s
```

分而治之

作为最后一个例子（你一定厌倦了易理解的代码，并想回到钻研比较抽象的规范的语言学讨论上来），让我们做一点超级计算。然后，休息。我们的超级计算应用程序是一个相当典型的并行处理模型（参见图 1-5）。我们有：

- 一台发生器，它产生可以并行执行的任务。
- 一组工人，用于处理任务。
- 一个接收器，用于收集工作进程返回的结果。

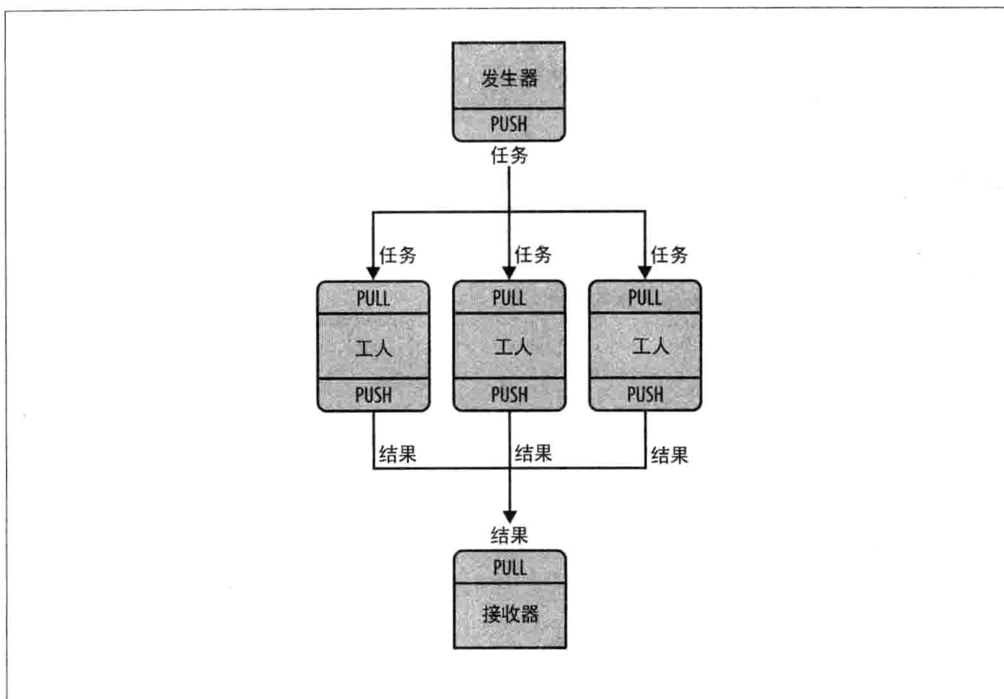


图1-5：并行流水线

在现实中，工人在超快的电脑上运行，可能会使用GPU（图形处理单元）来完成这个艰难的数学运算。示例1-8显示了用于发生器的代码。它产生100个任务，每个任务包含一个消息，用来告诉工人某个休眠所需的毫秒数。

示例1-8：并行任务发生器（taskvent.c） 17

```
//
//  任务发生器
//  将PUSH套接字绑定到tcp://localhost:5557
//  通过那个套接字发送批量任务给工人
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  用于发送消息的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_bind (sender, "tcp://*:5557");
```

```
    //  用于发送批次开始消息的套接字
    void *sink = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sink, "tcp://localhost:5558");

    printf ("Press Enter when the workers are ready:");
    getchar ();
    printf ("Sending tasks to workers...\n");

    //  第一个消息是 "0"，它表示批次的开始
    s_send (sink, "0");

    //  初始化随机数发生器
    srandom ((unsigned) time (NULL));

    //  发送 100 个任务
    int task_nbr;
    int total_msec = 0;     //  预期消耗的总毫秒数
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        int workload;
        //  从 1 到 100 毫秒的随机工作负载
        workload = randof (100) + 1;
        total_msec += workload;
        char string [10];
        sprintf (string, "%d", workload);
        s_send (sender, string);
    }
    printf ("Total expected cost: %d msec\n", total_msec);
    sleep (1);              //  给 ØMQ 时间来传递

    zmq_close (sink);
    zmq_close (sender);
    zmq_ctx_destroy (context);
    return 0;
}
```

18 用于工人应用程序的代码如示例 1-9 所示。它接收一个消息，按照消息里的那个秒数休眠，然后发出它完成任务的信号。

示例1-9：并行任务工人（taskwork.c）

```
//
//  任务工人
//  将 PULL 套接字连接到 tcp://localhost:5557
//  通过那个套接字收集来自发生器的工作负载
//  将 PUSH 套接字连接到 tcp://localhost:5558
```

```
//  通过那个套接字发送结果给接收器
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  用于接收消息的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  用于发送消息的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sender, "tcp://localhost:5558");

    //  永远地处理任务
    while (1) {
        char *string = s_recv (receiver);
        //  用于查看器的简易过程指示器
        fflush (stdout);
        printf ("%s.", string);

        //  不做工作
        s_sleep (atoi (string));
        free (string);

        //  将结果发送给接收器
        s_send (sender, "");
    }
       zmq_close (receiver);
    zmq_close (sender);
    zmq_ctx_destroy (context);
    return 0;
}
```

最后，示例 1-10 显示了接收器应用程序。它收集了 100 条消息，然后计算整体处理用了多长时间，所以我们可以证实，如果有一个以上的工人，它们确实是并行运行的。

示例1-10：并行任务接收器（tasksink.c） 19

```
//
//  任务接收器
//  将 PULL 套接字绑定到 tcp://localhost:5558
//  通过那个套接字收集来自各个工人的结果
//
```

```
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和套接字
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    //  等待批次的开始
    char *string = s_recv (receiver);
    free (string);

    //  现在启动时钟
    int64_t start_time = s_clock ();

    //  处理 100 个确认
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if ((task_nbr / 10) * 10 == task_nbr)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    //  计算并报告批次的用时
    printf ("Total elapsed time: %d msec\n",
        (int) (s_clock () - start_time));

    zmq_close (receiver);
    zmq_ctx_destroy (context);
    return 0;
}
```

一个批次的平均成本为 5 秒。当启动 1 个、2 个、4 个工人时，我们从接收器得到了如下的结果：

```
#   1 worker
Total elapsed time: 5034 msec
#   2 workers
Total elapsed time: 2421 msec
#   4 workers
Total elapsed time: 1018 msec
```

让我们来看看这段代码某些方面更多的详细信息： 20

- 工人向上连接到发生器，并且向下连接到接收器。这意味着你可以随意添加工人。如果工人绑定到其端点，每次添加一个工人，你需要（a）更多个端点和（b）修改发生器和 / 或接收器。我们说，发生器和接收器是架构的固定部分而工人是它的动态部分。
- 我们必须同步开始同批次所有工人的启动和运行。这是 ØMQ 中存在的一个相当普遍的疑难杂症，并没有简单的解决方案。connect 方法需要一定的时间，所以当一组工人连接到发生器时，第一个成功连接的工人会在这短短的时间得到消息的整个负载，而其他工人仍在进行连接。如果不知何故批次的开始不同步，那么系统就将无法并行运行。请尝试在发生器中移走等待，看看会发生什么。
- 发生器的 PUSH 套接字将任务均匀地分配给工人（假设批处理开始发出之前，它们都已连接）。这就是所谓的负载均衡，以后还会再关注它的更多细节。
- 接收器的 PULL 套接字均匀地收集来自工人的结果。这就是所谓的公平排队（参见图 1-6）。

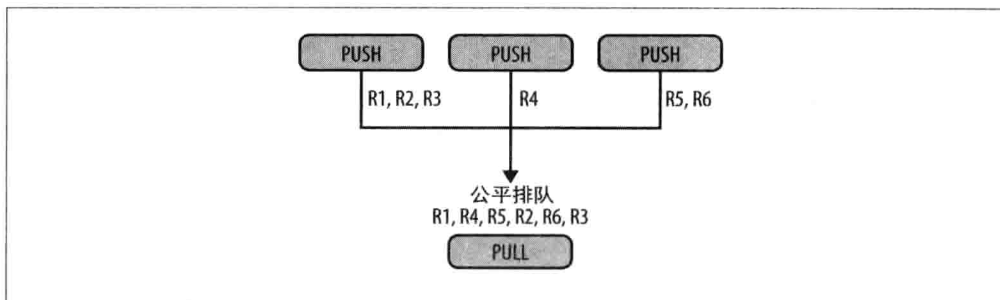


图1-6：公平排队

流水线模式也表现出“慢木匠”综合征，它导致了对 PUSH 套接字不能正确地负载均衡的指责。如果你使用的是 PUSH 和 PULL，并且你的某个工人得到比其他工人更多的信息，这是因为 PULL 套接字已比别人更快地连接，并在其他工人试图连接之前抓取了很多消息。

用 ØMQ 编程 21

看了这么多例子，你一定渴望开始在一些应用程序中使用 ØMQ。请在开始之前，深呼吸，冷静一下，并思考一些基本的建议，这将减少你很多的压力和困惑：

- 逐步了解 ØMQ。这只是一个简单的 API，但它隐藏了许多的可能性。慢慢地研究各

种可能性并掌握它们。

- 编写漂亮的代码。丑陋的代码隐藏了问题，并使得别人很难帮你。你可能会习惯用无意义的变量名，但阅读你的代码的人却不习惯。使用有意义的名称，而不要说“我太粗心，忘了告诉你这个变量的真正含义”。使用一致的缩进和干净的布局。编写漂亮的代码，你的世界就会更舒服。
- 在编程的同时做测试。当你的程序不能正常工作时，你应该知道哪 5 行代码是罪魁祸首。当你做 ØMQ 魔法时，情况尤其是这样，在最初几次尝试使用它时，它是不会工作的。
- 当你发现代码未如预期般运作时，请将你的代码分解成片段，测试每一个片段，看看哪一个不工作。ØMQ 让你可以编写本质上模块化的代码，用它来发挥你的优势。
- 在需要的时候使用抽象（类、方法，等等）。如果你复制 / 粘贴大量的代码，你也将复制 / 粘贴错误。

获取正确的上下文

ØMQ 应用程序总是从创建一个上下文开始，然后使用它来创建套接字。在 C 语言中，它是 zmq_ctx_new() 调用。你应该在你的进程中只创建并使用一个上下文。从技术上讲，上下文是指在单个进程中所有套接字的容器，并充当 inproc 套接字的传输工具，这是在一个进程中连接线程的最快方法。如果在运行时，某个进程有两个上下文，那么这些都像独立的 ØMQ 实例。如果这是你明确想要的，那也没什么，但在其他方面要记住：

在你的主代码的开始处执行一个 zmq_ctx_new()，在代码最后执行一个 zmq_ctx_destroy()。

如果你正在使用 fork() 系统调用，那么每个进程都需要自己的上下文。如果你在主进程中先执行 zmq_ctx_new()，再调用 fork()，子进程就会获得自己的上下文。一般来说，你想在子进程中做有趣的东西，而在父进程中只管理这些子进程。

22 执行彻底的退出

优雅的程序员与优雅的职业杀手共享相同的座右铭：始终在完成工作时执行清理。当你在像 Python 的语言中使用 ØMQ 时，内存会自动释放。但是，在采用 C 语言时，你必须在完成对象的处理后仔细地释放它们，否则就会造成内存泄漏、不稳定的程序，以及坏的结果。

内存泄漏是一回事，但 ØMQ 对于你怎么退出一个应用程序是非常讲究的。其原因是技术性的，但结果是，如果你让任何套接字保持打开，zmq_ctx_destroy() 函数将一直挂起。

即使你关闭所有套接字，如果有悬而未决的连接或发送，zmq_ctx_destroy()默认也将等待下去，除非你在关闭这些套接字之前，将它们的 LINGER（延期）设置为零。

我们需要担心的 ØMQ 对象包括消息、套接字和上下文。幸运的是，至少在简单的程序中，这是相当简单的：

- 处理完消息的那一刻，总是用 zmq_msg_close()关闭它。
- 如果你打开和关闭了很多套接字，这可能标志着你需要重新设计你的应用程序。
- 当你退出程序时，关闭你的套接字，然后调用 zmq_ctx_destroy()。这销毁了上下文。

至少 C 语言开发的情况是这样的。在自动销毁对象的语言中，当你离开作用域时，套接字和上下文将予以销毁。如果你使用异常，必须在一个类似“final”块的东西中执行清理，这对任何资源都是一样的。

如果你正在做多线程工作，情况就会比这更复杂。我们将在下一章研究多线程，但是由于有些人会不顾警告，尝试在可以安全地走之前跑，下面是一个快速和难看的指导，告诉他们如何在多线程 ØMQ 应用程序中执行彻底的退出。

首先，不要尝试在多个线程中使用同一个套接字。请不要解释为什么你认为这将是非常有趣的，只要记住别这样做。接下来，你需要关闭每个有持续请求的套接字。正确的方法是先设置一个低的 LINGER 值（1 秒），然后关闭套接字。如果当你销毁一个上下文时，你绑定的语言不自动为你做这个工作，我建议发送补丁。

最后，销毁上下文。这将导致任何连接到线程（即，它们共享相同的上下文）的阻塞的接收或调查或发送都返回一个错误。捕获该错误，然后在该线程中设置 LINGER，关闭套接字并退出。不要多次销毁同样的上下文。在主线程中的 zmq_ctx_destroy()调用将会保持阻塞，直到它知道的所有套接字都已安全地关闭为止。

瞧！这是多么复杂和令人痛苦的，所以任何语言绑定程序的称职作者都会自动执行此操作，使得用户不必再去关闭套接字。 23

为什么我们需要 ØMQ

现在你已经在实践中见过了 ØMQ，让我们回到“为什么”。

当今的许多应用程序都包含了跨越某种网络的组件，无论这种网络是局域网还是互联网。因此，许多应用程序开发者最终都会处理某种类型的消息传递。一些开发人员使用消息队列产品，但大多数时候，他们使用 TCP 或 UDP 自己做。这些协议并不难用，但是，从 A 发送几个字节到 B 和以任何一种可靠的方式处理消息，这两者之间有很大的区别。

让我们来看看当开始使用原始的 TCP 连接部件的时候，我们要面对的典型问题。任何可复用的消息层都需要解决如下所有这些问题或其中的大部分问题：

- 我们如何处理 I/O 呢？是让我们的应用程序阻塞，还是在后台处理 I/O 呢？这是一个关键的设计决策。阻塞式 I/O 创建的架构不能很好地扩展，但后台 I/O 也是非常难以正确做到的。
- 我们如何处理动态组件（例如，暂时撤除的块）呢？我们需要正式将组件划分为“客户端”和“服务器”，并强制该服务器不能撤除吗？那么，如果我们想将服务器连接到服务器时该怎么办呢？我们需要每隔几秒钟就尝试重新连接吗？
- 我们如何表示在线路上的消息呢？我们应该怎样将数据组织为帧，才能使得它很容易写入和读取，避免缓冲区溢出，既对小型消息高效，也足以处理非常大的戴着聚会礼帽的跳舞猫的视频呢？
- 我们如何处理不能立即传递的消息呢？特别是当我们在等待一个组件的联机回应时如何处理呢？我们需要丢弃消息，把它们放入一个数据库，或者把它们放到一个内存队列吗？
- 我们在哪里存储消息队列呢？如果组件从队列中读取很慢，导致我们的队列堆积，这会发生什么情况？我们的策略是什么呢？
- 我们如何处理丢失的消息呢？我们应该等待新的数据，要求重发，还是应该建立某种可靠性层，确保信息不会丢失呢？如果该层本身崩溃了该怎么办呢？
- 如果我们需要使用一个不同的网络传输，比如说，用多播来取代 TCP 单播，或 IPv6，该怎么办呢？我们需要重写应用程序吗？还是将传输抽象到某个层中呢？
- 我们如何路由消息呢？我们可以发送同样的消息到多个接收者吗？我们可以发送应答给原来的请求者吗？
- 24 我们如何编写出另一种语言的 API 呢？我们应该重新实现一个线路级协议，还是重新包装一个库？如果是前者，我们怎么能保证协议栈的高效稳定呢？如果是后者，我们又怎么能保证互操作性呢？
- 我们应该如何表示数据，以便它可以在不同的架构之间读取呢？我们应该对数据类型强制执行特定的编码吗？究竟到什么程度，才是消息传递系统的工作，而不是更高一层的工作呢？
- 我们应该如何处理网络错误呢？是等待并重试，默默地忽略它们，还是终止它们呢？

以 Apache ZooKeeper（*http://zookeeper.apache.org*）这个典型的开源项目为例，请阅读在 *src/c/src/zookeeper.c* 中的 C API 代码（*http://github.com/apache/zookeeper/blob/trunk/src/c/src/zookeeper.c*）。当我在 2010 年阅读这段代码时，它是 3200 行的神秘代码，并且包含一个未在文档中记录的客户端 / 服务器网络通信协议。我发现，它的高效是因为它使用的是 `poll()` 而不是 `select()`。不过说真的，ZooKeeper 应使用一个通用的消息层

和一个在文档中明确记载的线路层协议。对开发团队来说，重复制造特殊的轮子，这是令人难以置信的浪费。

但是，我们如何才能做一个可复用的消息传递层呢？为什么有那么多的项目都需要这项技术，人们都还在通过在他们的代码中驱动 TCP 套接字费力地做它，并重复解决一长串清单中的问题呢（参见图 1-7）？

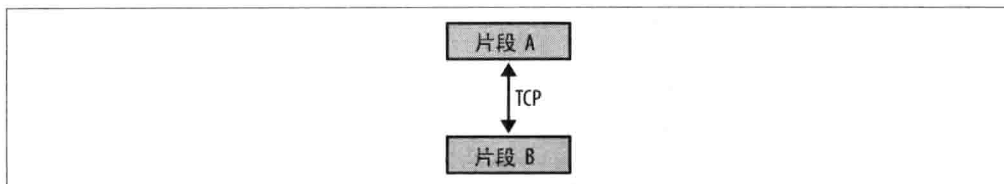


图1-7：开始阶段的消息传递

事实证明，构建可复用的消息传递系统是非常困难的，这就是为什么很少有自由和开放源码（FOSS）项目尝试做这项工作的原因，它也是商业通信产品复杂、昂贵、灵活性差，而且脆弱的原因。2006 年，iMatix 设计了高级消息队列协议，或 AMQP（*http://www.amqp.org*），开始给 FOSS 开发者提供也许是首个可复用的消息传递系统方法。AMQP 工作得比许多其他设计更好，但仍然相对复杂、昂贵，而且脆弱（*http://www.imatix.com/articles:whats-wrong-with-amqp*）。学会如何使用它需要几个星期，而要用它来建立在事情变得很麻烦时也不会崩溃的稳定架构则需要几个月。

大多数消息传递项目（如 AMQP）都在尝试以可重用的方式解决上面这个冗长的清单上的问题，它们通过发明一种负责寻址、路由和排队的新概念——代理，来做到这一点。这将导致一个客户端 / 服务器协议或一些未在文档中记录的协议之上的一组 API，它允许应用程序与这个代理交流。在降低大型网络的复杂性方面，代理是一个很好的东西。 25
但把以代理为基础的消息传递添加到像 ZooKeeper 这样的产品会使情况变得更糟，而不是更好。这将意味着增加一台额外的大电脑和一个新的单点故障。代理迅速成为一个瓶颈和一个要管理的新风险。如果软件支持的话，我们可以添加第二个、第三个和第四个代理，并提出一些故障切换方案。人们这么做了。然而，它产生了更多的变动部件，变得更复杂，有更多的东西会被破坏。

此外，以代理为中心的设置都需要自己的运营团队。你真的需要日夜注意代理，并在它们开始“行为不端”时，用棍子敲打它们。你需要电脑，你需要备份的电脑，你需要人来管理那些电脑。只有那些由多个团队在数年内建成的，带有许多变动部件的大型应用程序，才值得这样做。

因此，中小型应用程序开发人员陷入了困境。他们要么避免网络编程，制作不可扩展的单一应用程序，要么跳进网络编程，制作脆弱、复杂、很难维护的应用程序。他们还可以把赌注压在消息传递产品上，并最终获得依赖于昂贵且易破坏的技术的可扩展的应用程序。目前还没有非常好的选择，这也许可以解释为什么消息传递主要还停留在上个世纪，并激起强烈的情绪——对用户是消极的，而对那些销售技术支持和许可的厂商则是欢乐的、喜悦的（参见图 1-8）。

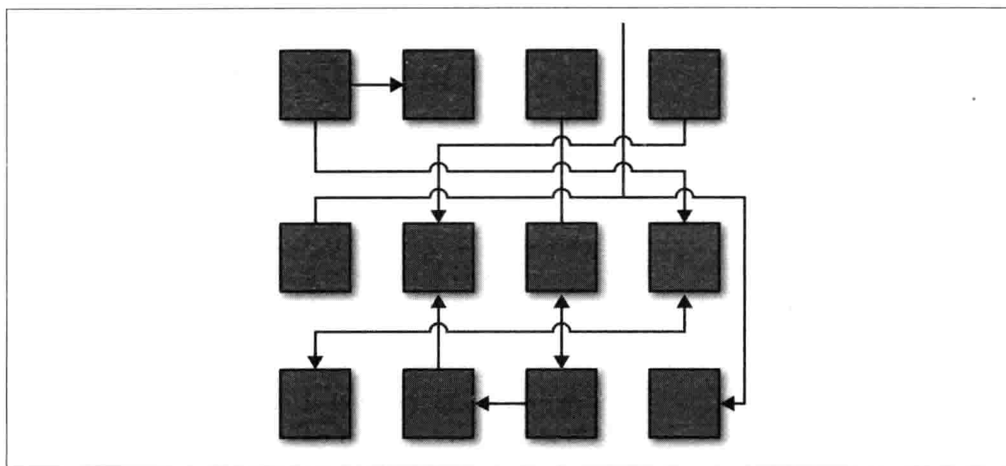


图1-8：消息传递的演变

我们需要的是做消息传递工作的东西，但需要它以下面这种简单和廉价的方式完成工作，它可以在任何应用程序中以接近零的消耗开展工作。它应该是不需要任何其他依赖就可
26 以链接的库。无须额外的变动部件，所以没有额外的风险。它应该能运行在任何操作系统上，并能用任何编程语言开展工作。

而这就是 ØMQ：一个高效的可嵌入库，它解决了大部分应用程序需要解决的问题，变得在网络上有良好的可伸缩性，而没有多少成本。

具体做法是：

- 它在后台线程异步处理 I/O。这些线程使用无锁数据结构与应用程序线程进行通信，所以并发 ØMQ 应用程序不再需要锁、信号量，或其他等待状态。
- 组件可以动态地来去自如，而 ØMQ 会自动重新连接。这意味着你可以以任何顺序启动组件。你可以创建“面向服务的架构”（SOA），其中的服务可以在任何时间加入和离开网络。
- 它根据需要自动对消息排队。为此，它会智能地在对消息排队之前，将消息尽可能

地推进到接收者。

- 它有一个处理过满队列（称为“高水位标志”）的方法。当队列满时，ØMQ 会自动阻止发件人，或丢弃消息，这取决于你正在做的是哪种消息传递（即所谓的“模式”）。
- 它可以让你的应用程序通过任意传输协议来互相交流，这些协议可以是：TCP、多播、进程内、进程间。你不需要更改代码以使用不同的传输工具。
- 它使用依赖于消息传递模式的不同策略，安全地处理速度慢 / 阻塞的读取者。
- 它可以让你采用多种模式，如请求 - 应答和发布 - 订阅来将消息路由。这些模式是指你如何创建拓扑结构和网络结构。
- 它可以让你创建代理（proxy）来排队、转发，或通过一个调用来捕获消息。代理可以降低网络互联的复杂性。
- 它使用在线路上的简单组帧原封不动地传递整个消息。如果你写了一个 10KB 的消息，那么你将收到一个 10KB 的消息。
- 它不对消息强加任何格式。它们是零字节到千兆字节的二进制大对象。当你想表示你的数据时，可以选择其上的其他一些产品，如谷歌的协议缓冲区、XDR 等。
- 它能智能地处理网络错误。有时候它会重试，有时它会告诉你某个操作失败。
- 它可以减少你的能源消耗。少花 CPU 多办事意味着使用电脑更少的能源，你可以让你的旧电脑使用更长的时间。戈尔（译者注：美国前副总统，环保主义者）也会爱上 ØMQ 的。

实际上，ØMQ 做的比这更多。它对你如何开发网络功能的应用程序有颠覆性的影响。 27
从表面上看，这是一个在其上做 `zmq_msg_recv()` 和 `zmq_msg_send()` 的套接字风格的 API。但该消息处理循环迅速成为中心循环，而你的应用程序很快就会分解成一组消息处理任务。它是优雅和自然的。而且，它可扩展：每个任务对应一个节点，节点通过任意传输方式互相交谈。在一个进程中的两个节点（节点是一个线程），在一台电脑中的两个节点（节点是一个进程），或一个网络上的两台电脑（节点是一台电脑），所有的处理方式都是相同的，不需要更改应用程序代码。

套接字的可扩展性

让我们来看看 ØMQ 在实践中的可扩展性。下面是一个 shell 脚本，它启动天气服务器，然后并行启动一批客户端：

```
wuserver &
wuclient 12345 &
wuclient 23456 &
wuclient 34567 &
wuclient 45678 &
wuclient 56789 &
```

作为客户端运行，我们使用 top 来看看活动的进程，我们（在一台四核电脑）看到如下的内容：

```
PID USER      PR  NI  VIRT   RES  SHR  S  %CPU %MEM    TIME+    COMMAND
7136 ph       20   0  1040m  959m 1156 R  157  12.0   16:25.47  wuserver
7966 ph       20   0  98608  1804 1372 S   33   0.0    0:03.94  wuclient
7963 ph       20   0  33116  1748 1372 S   14   0.0    0:00.76  wuclient
7965 ph       20   0  33116  1784 1372 S    6   0.0    0:00.47  wuclient
7964 ph       20   0  33116  1788 1372 S    5   0.0    0:00.25  wuclient
7967 ph       20   0  33072  1740 1372 S    5   0.0    0:00.35  wuclient
```

让我们思考一下这里发生了什么。天气服务器只有一个套接字，但在这里我们让它并行发送数据到 5 个客户端。我们可以有成千上万的并发客户端。服务器应用程序看不到它们，也不直接跟它们交流。所以 ØMQ 套接字像一个小型服务器，它默默接受客户端请求，并将数据以尽可能与网络处理一样快的速度推送出去。这是一个多线程的服务器，它能从你的 CPU 压榨出更多的处理能力。

从 ØMQ v2.2 升级到 ØMQ v3.2

在 2012 年初，ØMQ v3.2 变得足够稳定，可供现场使用，而且当你正在读这本书的时候，它应该是你真正使用的版本。如果你还在使用 2.2 版，下面是关于功能修改和对如何迁移你的代码的一个简单的总结性说明。

28 发布 - 订阅过滤如今在发布方，而不是在订阅方完成。这在许多发布 - 订阅用例中显著提高了性能。你可以放心地混搭使用 3.2 版和 2.1/2.2 版的发布者和订阅者。

大部分的 API 都是向下兼容的，只有一小部分进入 v3.0 的变化没有考虑到破坏现有代码的成本。zmq_send() 和 zmq_recv() 的语法改变了，并且 ZMQ_NOBLOCK 被重新命名为 ZMQ_DONTWAIT。所以，虽然我很想说，“你只要用最新的 libzmq 重新编译你的代码就可以了”，但实际情况不是那样的。无论如何，我们以后会禁止这种 API 的破坏。

因此，对于使用低级别 libzmq API 的 C/C++ 应用程序，最低限度的修改是将所有对 zmq_send() 的调用替换为调用 zmq_msg_send()，并将 zmq_recv() 替换为 zmq_msg_recv()。在其他语言中，你的绑定作者可能已经完成了这项工作。注意，这两个函数现在在出错的情况下返回 -1，并根据有多少字节被发送或接收返回零或更大的值。

该 libzmq API 的其他部分变得更加一致。我们不推荐使用 zmq_init() 和 zmq_term()，而用 zmq_ctx_new() 和 zmq_ctx_destroy() 来取代它们。我们增加了 zmq_ctx_set()，让你在开始用它来工作之前配置一个上下文。

最后，我们通过 zmq_ctx_set_monitor() 调用添加对上下文的监测，它可以让你追踪连接和断开连接，以及其他套接字事件。

警告：不稳定的典范！

传统的网络编程建立在“一个套接字与一个连接、一个接收者会话”的一般假定上。虽然有各种多播协议，但这些都是奇特的。当我们假设“一个套接字 = 一个连接”时，我们在用特定方式扩展我们的架构。我们创建逻辑线程，其中每个线程都可以与一个套接字、一个节点打交道。我们把智能和状态都放在这些线程中。

在 ØMQ 的世界中，套接字是通向自动为你管理一整套连接的快速小型后台通信引擎的入口的。你无法看到、处理、打开、关闭或附加状态到这些连接。无论你使用阻塞的发送还是接收或轮询，你可以交流的是套接字，而不是它为你管理的连接。连接是私有的和看不见的，这就是 ØMQ 的可扩展性的关键。

因为你的代码在跟一个套接字交流，所以它就可以不加修改地跨越任何网络协议处理任意数量的连接。基于 ØMQ 的消息传递模式与基于你的应用程序代码的消息传递模式相比，它可以更廉价地扩展。

所以，一般的假设不再适用。当你阅读代码示例时，你的大脑会尝试将它们联系到你所知道的知识。你会读到“套接字”，并认为“啊，那代表到另一个节点的连接”。但那是不对的。你会读到“线程”，而你的大脑就会重新思考，“啊，一个线程代表到另一个节点的连接”，而结果是你的大脑再次犯了错误。

29

如果你是第一次阅读这本书，请意识到，除非你真正花一两天（或者三四天）编写 ØMQ 代码，否则你可能会感到困惑，尤其会对 ØMQ 让你的事情变得多么简单感到困惑，你可以尝试将一般假设强加给 ØMQ，而它不会工作。然后当这一切变得清晰时，那“稀里哗啦”的根本变化的顿悟时刻，就是你体会到启蒙和信任的时刻。

第 2 章

套接字和模式

在第 1 章中，我们用 ØMQ 作为一个驱动，使用一些基本的例子介绍了主要的几种 ØMQ 模式：请求 - 应答、发布 - 订阅和流水线。在本章中，我们将开始实践，并开始学习如何在实际程序中使用这些工具。

我们将讨论：

- 如何创建 ØMQ 套接字和使用它来工作
- 如何发送和接收套接字上的消息
- 如何围绕 ØMQ 的异步 I/O 模型构建应用程序
- 如何在一个线程中处理多个套接字
- 如何妥善处理致命性和非致命错误
- 如何处理如 Ctrl-C 中断信号
- 如何彻底地关闭 ØMQ 应用程序
- 如何检查 ØMQ 应用程序是否发生内存泄漏
- 如何发送和接收多部分消息
- 如何在网络上转发消息
- 如何构建一个简单的消息队列代理
- 如何使用 ØMQ 编写多线程应用程序
- 如何使用 ØMQ 在线程之间发信号
- 如何使用 ØMQ 协调多个节点组成的网络
- 如何创建和使用发布 - 订阅的消息封装
- 如何使用高水位标记（HWM）来防止内存溢出

32

套接字 API

坦白地说，ØMQ 对你来说确实是一种诱饵游戏，对此我们不怀歉意。这是为你自己好，它对我们的伤害超过了对你的伤害。它提出了一个熟悉的基于 socket 的 API，这隐藏了一堆消息处理引擎，对此我们做了大量的工作。但是，这个结果会慢慢改变你如何设计和编写分布式软件的观念。

套接字是网络编程事实上的标准 API，对让你停止哭泣也很有用。一件使开发人员觉得 ØMQ 特别合胃口的事情是，它使用套接字和消息而不是其他的任意一组概念。感谢 Martin Sustrik 的这一创举。它将原来保证让整个机房人得紧张症的"面向消息的中间件"变为一个短语"特辣套接字"，这使我们对"比萨饼"充满奇怪的渴望并有了对其了解更多的欲望。

就像一道最喜欢的菜，ØMQ 套接字很容易消化。这些套接字有 4 个部分的生存期，就像 BSD 套接字：

- 我们可以创建和销毁它们，这共同形成套接字的生命周期（参见 zmq_socket() 和 mq_close()）。
- 我们可以通过为它们设置选项和进行必要的检查来配置它们（参见 zmq_setsockopt() 和 zmq_getsockopt()）。
- 我们可以通过创建进出它们的 ØMQ 连接，将它们插入到网络拓扑结构中（参见 zmq_bind() 和 zmq_connect()）。
- 我们可以通过在它们上面写入和接收这些消息来传递数据（参见 zmq_msg_send() 和 zmq_msg_recv()）。

需要注意的是，套接字总是无类型指针，而消息（我们将很快论及它们）是结构。因此，在 C 语言里，你按原样传递套接字，但将消息地址传递给所有处理消息的函数，比如 zmq_msg_send() 和 zmq_msg_recv()。请记住并认识到"在 ØMQ 中，所有的套接字都属于我们"，但消息是你在代码中真正拥有的东西。

对于任何对象，创建、销毁并配置套接字都会如你所愿地工作。但请记住，ØMQ 是异步的，可伸缩的。这对我们如何将套接字插入到网络拓扑结构，以及随后我们如何使用套接字会有一些影响。

把套接字接入网络拓扑

为了在两个节点之间创建连接，你可以在一个节点中使用 zmq_bind()，并在另一个节点中使用 zmq_connect()。按照通常的经验法则，执行 zmq_bind() 的节点是一台"服务器"，

它有一个公认的网络地址，而执行 zmq_connect() 的节点是一台“客户端”，它的地址要么是未知的，要么是任意的。因此，我们说我们“把一个套接字绑定到一个端点”和“把一个套接字连接到一个终端”，终端是指公认的网络地址。 33

ØMQ 的连接与旧式的 TCP 连接有些不同。主要的明显不同是：

- 它们跨任意的传输协议（inproc、ipc、tcp、pgm 或 epgm)。参见 zmq_inproc()、zmq_ipc()、zmq_tcp()、zmq_pgm() 和 zmq_epgm()。
- 一个套接字可能会有很多输入和输出的连接。
- 不存在 zmq_accept() 方法，当一个套接字被绑定到一个端点的时候，它自动地开始接受连接。
- 网络连接本身是在后台发生的，而如果网络连接断开(例如，对等节点消失后又回来)，ØMQ 会自动地重新连接。
- 应用程序不能与这些连接直接交流，它们是被封装在套接字之下的。

很多架构都采用客户端 - 服务器模式，其中服务器通常是静态的部分，客户端通常是动态的部分，即它们经常会加入或者离开。有一些寻址方面的问题：服务器对客户端是可见的，但是客户端对服务器却未必可见。因此，通常哪个节点应该执行 zmq_bind()（服务器）和哪个节点应该执行 zmq_connect()（客户端）是明显的。它也取决于你使用的套接字的类型，不寻常的网络架构例外。后面我们将会研究套接字类型。

现在假设我们在启动服务器前（*before*）启动客户端。在传统的网络中，我们会得到一个大的红色的失败标志。但 ØMQ 允许我们任意地启动和停止各部件。一旦客户端节点执行 zmq_connect()，连接就建立起来了，而该节点就能够开始把消息写入套接字中。在某个时候（希望是在因排队消息太多，消息被丢弃或者客户端阻塞之前）服务器开始工作，执行 zmq_bind()，而 ØMQ 开始分发消息。

一个服务器节点可以绑定到很多端点（即协议和地址的组合），而只需要单个套接字就能够做到。这就意味着它能接受不同的传输协议：

```
zmq_bind (socket, "tcp://*:5555");
zmq_bind (socket, "tcp://*:9999");
zmq_bind (socket, "inproc://somename");
```

对于大多数传输协议，你不能两次绑定到相同的端点（例如：不像在 UDP 中）。但是，ipc 传输协议确实允许一个进程绑定到已被其他进程使用的端点。这意味着允许一个进程在崩溃后恢复。

虽然 ØMQ 试图对哪一端绑定和哪一端连接保持中立，但两端还是有区别的。随后我们可以看到这方面的更详细的介绍。这就意味着你应该总是认为“服务器”是拓扑中的固 34

定部分，有相对固定的终端地址，而“客户端”作为可以动态加入和撤出的部分。然后以这种模式设计你的应用程序，它运行起来很可能会更好。

套接字具有类型。套接字的类型定义了套接字的意义：其向内或向外路由消息、排队等策略。你可以把特定类型的套接字连接在一起，例如，把一个发布者类型的套接字和一个订阅者类型的套接字连接在一起。套接字以“消息模式”的形式协同工作。随后我们可以看到它的更详细的介绍。

以不同的方式连接套接字的能力让 ØMQ 有了作为消息队列的基础。在这上面有各个层，例如代理，我们随后会讨论到。但本质上，ØMQ 通过把各部分连接在一起的方式来定义你的网络架构，就像一个孩子拼接他的建筑玩具一样。

使用套接字来传输数据

要发送和接收消息，应使用 `zmq_msg_send()` 和 `zmq_msg_recv()` 方法。该名称是常规的，但 ØMQ 的 I/O 模型与 TCP 模型有很大区别（参见图 2-1），你需要时间来转变观念。

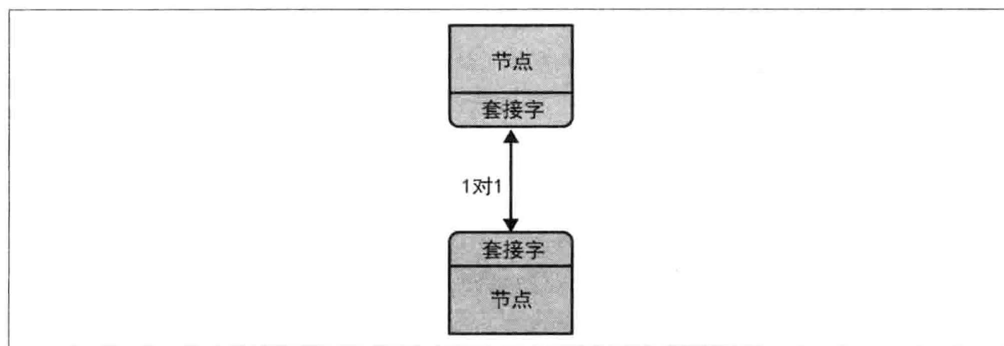


图2-1：TCP套接字是1对1的

让我们来看看当涉及处理数据时，TCP 套接字和 ØMQ 套接字之间的主要差异：

- ØMQ 套接字像 UDP 那样传递信息，而不是像 TCP 那样传递字节流。ØMQ 消息是指定长度的二进制数据。我们不久就会来研究消息，因为它们的设计针对性能进行了优化，所以有点棘手。
35 - ØMQ 套接字在一个后台线程执行自己的 I/O。这意味着消息到达本地输入队列并从本地输出队列被发送，不管你的应用程序正忙着做什么。
- ØMQ 套接字根据套接字类型具有内置的 1 对 N 的路由行为。

zmq_msg_send() 方法实际上并没有将消息发送到套接字连接。它会将消息排队，这样 I/O 线程可以将其异步发送。除了在某些异常情况下，它不会阻塞。因此，当 zmq_msg_send() 返回你的应用程序时，该消息并不一定被发送。

单播传输

ØMQ 提供了一组单播传输（inproc、ipc 和 tcp）和多播传输（epgm、pgm）。多播是一种先进的技术，我们将在后面研究它。除非你知道你的扇出率将使得 1 对 N 的单播是不可能的，否则甚至都不要开始使用它。

在最常见的情况下，使用 tcp，这是一个断开连接（*disconnected*）的 TCP 传输。它是可伸缩、可移植的，并且在大多数情况下足够快。我们之所以称之为“断开连接”，是因为 ØMQ 的 TCP 传输在你连接到端点前，不需要端点存在。客户端和服务器可以在任何时候连接和绑定，可以出去和回来，并且它对应用程序保持透明。

进程间的 ipc 传输也是断开连接的。有一个限制：它不能在 Windows 上工作。按照惯例，我们使用“.ipc”扩展名为端点命名，以避免与其他文件名有产生冲突的可能。在 UNIX 系统中，如果你使用 ipc 端点，需要使用适当权限创建这些，否则，它们可能无法在不同用户 ID 运行的进程之间共享。你还必须确保所有的进程都可以访问该文件，例如，通过在同一个工作目录下运行。

在线程间的传输中，inproc 是一个连接的信令传送，它比 tcp 或 ipc 快得多。与 tcp 和 ipc 相比，这种传输有特定的限制，*但服务器必须在任何客户端发出连接之前，发出一个绑定请求*。ØMQ 的未来版本可能会解决这个问题，但目前这定义了你如何使用 inproc 套接字。我们创建和绑定一个套接字，并启动子线程，从而创建并连接其他套接字。

ØMQ 不是一个中性载体

ØMQ 初学者常问的一个问题（这是一个我曾问过我自己的问题）是，“我怎样用 ØMQ 编写一个某某服务器呢？”举个例子，“我怎样用 ØMQ 编写一个 HTTP 服务器呢？”言下之意是，如果我们能使用普通套接字承载 HTTP 请求和响应，那么应该能够使用 ØMQ 套接字做同样的事情，而且只会做得更快，更好。

答案曾经是，“这不是它的工作方式。”ØMQ 不是一个中性载体，它规定了它使用的传 36
输协议的框架。这个框架不与现有的往往用自己的框架的协议兼容。例如，比较一个 HTTP 请求（参见图 2-2）和一个 ØMQ 请求（参见图 2-3），两个都在 TCP/IP 上运行。HTTP 请求使用 CRLF（回车换行）作为其最简单的框架分隔符，而 ØMQ 使用指定了长度的帧。

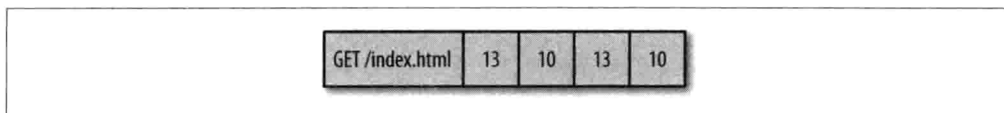


图2-2：HTTP在线路上

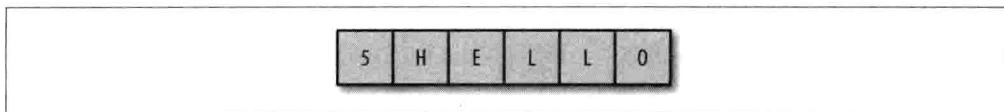


图2-3：ØMQ在线路上

所以，你可以使用 ØMQ 编写类似于 HTTP 的协议，例如使用请求 - 应答套接字模式来实现。但它不会是 HTTP。

然而，从 ØMQ v3.3 开始，ØMQ 有了一个叫作 ZMQ_ROUTER_RAW 的套接字选项，让你无须利用 ØMQ 帧来读取和写入数据。你可以使用它来读取和写入正确的 HTTP 请求和响应。Hardeep Singh 促成了这个变化，以使他可以从他的 ØMQ 应用程序连接到远程登录服务器。在写作的时候，这个选项仍处于实验阶段，但它显示了 ØMQ 是如何不断发展以解决新问题的。也许下一个补丁就是你贡献的。

I/O 线程

我们说过，ØMQ 在一个后台线程中执行 I/O。除最极端的应用程序外，一个 I/O 线程（用于所有套接字）足以供所有应用程序使用。当你创建一个新的上下文时，它从一个 I/O 线程开始。一般的经验法则是，让每秒的每 GB 的输入或输出数据使用一个 I/O 线程。为了提高 I/O 线程的数量，可在创建任何套接字前使用 zmq_ctx_set() 调用：

```
int io-threads = 4;
void *context = zmq_ctx_new ();
zmq_ctx_set (context, ZMQ_IO_THREADS, io_threads);
assert (zmq_ctx_get (context, ZMQ_IO_THREADS) == io_threads);
```

我们已经看到，一个套接字一次可以处理几十个，甚至数千个连接。这对你如何编写应用程序有根本性的影响。传统的网络应用程序每个远程连接都有一个进程或一个线程，
37 并且该进程或线程处理一个套接字。ØMQ 允许你把这整个结构收缩到单个进程中，然后根据扩展的需要再拆散它。

如果你仅把 ØMQ 用于线程间通信（即一个不执行外部套接字 I/O 的多线程的应用程序），可以把 I/O 线程数设置为零。但这不是一个明显的优化，更多的是好奇心。

消息传递模式

在 ØMQ 的套接字 API 的牛皮纸包装下面是消息传递模式的世界。如果你有企业级消息传递的开发背景，或者对 UDP 有良好的理解，那么对这些将会比较熟悉。但对于 ØMQ 的初学者来说，它们令人惊讶，因为我们习惯了套接字一一映射到另一个节点的 TCP 模式。

让我们概括一下 ØMQ 所做的工作。它快速而高效地把整块数据（消息）发送到节点，这里的节点可以是线程、进程或节点。ØMQ 给你的应用程序提供一个单独套接字 API 来开展工作，而不管实际使用的传输协议是什么（例如，进程内、进程间、TCP 或多播）。各对等节点撤销或者接入的时候，ØMQ 套接字都会自动重新连接。它会根据需要将消息同时在发送者和接收者处进行排队。它仔细地管理这些队列，以确保进程不会耗尽内存而在适当的时候溢出到磁盘。它会处理套接字错误，在后台线程执行所有的 I/O，并采用无锁技术在节点之间进行会话，所以永远不会有锁定、等待、信号量或死锁问题。

但是，洞穿这一点，它根据称为*模式*（*pattern*）的精确攻略对消息进行路由和排队。正是这些模式为 ØMQ 提供了智能。它们封装我们辛苦得来的分配数据和工作的最佳途径的经验。ØMQ 的模式是硬编码的，但未来的版本可能会允许用户自定义模式。

ØMQ 模式是由类型相匹配的套接字对来实现的。换句话说，为了理解 ØMQ 模型，你要先理解套接字类型以及它们是如何协同工作的。大多数情况下，要做到这些只有通过学习，因为在这个层次，几乎没什么是显而易见的。

内置的核心 ØMQ 模式是：

- *请求 - 应答*（*request-reply*）模式，它将一组客户端连接到一组服务。这是一个远程过程调用和任务分发模式。
- *发布 - 订阅*（*publish-subscribe*）模式，它将一组发布者连接到一组订阅者。这是一个数据分布模式。
- *管道*（*pipeline*）模式，它以一种扇出 / 扇入模式连接各节点，其中可以有多个步骤和循环。这是一个并行任务分发和收集模式。

我们已经在第 1 章研究了这些模式。当人们仍然以传统的 TCP 套接字角度看待 ØMQ 时，还有另外一个他们倾向于尝试使用的模式：*独占对*（*exclusive pair*），它独占地连接两个套接字。这个模式只应该在连接进程中的两个线程时使用。在本章的最后，我们会看到 38
它的一个例子。

`zmq_socket()` 手册页相当清楚地说明了各种模式，需要阅读好几遍才能了解它的意义。下面这些套接字组合是有效的连接 - 绑定对（任何一方都可以绑定）：

- PUB 和 SUB
- REQ 和 REP
- REQ 和 ROUTER
- DEALER 和 REP
- DEALER 和 ROUTER
- DEALER 和 DEALER
- ROUTER 和 ROUTER
- PUSH 和 PULL
- PAIR 和 PAIR

你还可以看到对 XPUB 和 XSUB 套接字的引用，我们以后会来研究它（它们像 PUB 和 SUB 的原始版本）。任何其他组合都会产生未在文档中说明且不可靠的结果，在 ØMQ 的未来版本中，如果你尝试使用它们，就可能会返回错误。当然，你可以通过代码来桥接其他套接字类型（例如，从一个套接字类型读取并写入另一个套接字类型）。

高级别消息传递模式

上述 4 个核心模式被预置进了 ØMQ 中，它们是用核心 C ++ 库实现的 ØMQ API 的一部分，并保证可以在一切优秀的零售商店中获得。

在这些基础上，我们添加各种高级别的模式（*high-level pattern*）。我们在 ØMQ 之上构建这些高级别的模式，并用正在应用程序中使用的任何一种语言来实现它们。它们不是核心库的一部分，不随 ØMQ 包配备，并在自己的空间作为 ØMQ 社区的一部分存在。例如，我们将在第 4 章探讨的管家模式，它驻留在 ØMQ 组织的 GitHub 的管家（Majordomo）项目中。

本书的目标之一是为读者提供一组这样的高级别的模式，无论是小型的（如何明智地处理消息）还是大型的（如何制作一个可靠的发布 - 订阅架构）。

39

处理消息

在线路上，ØMQ 消息是大小从零向上，适合在内存中容纳的任意的块。你使用 protobufs、msgpack、JSON，或任何其他应用程序需要用来会话的东西来做你自己的序列化。明智的做法是选择一个便携和快速的数据表示，但关于权衡你可以有自己的决策。

在内存中，ØMQ 消息是 zmq_msg_t 结构（或类，这取决于你采用的语言）。下面是在 C 语言中使用 ØMQ 消息的基本规则：

- 创建并传递 zmq_msg_t 对象，而不是数据块。

- 要读取消息，可使用 zmq_msg_init() 来创建一个空的消息，然后将其传递到 zmq_msg_recv()。
- 要把新数据写入消息，则使用 zmq_msg_init_size() 来创建消息，并在同一时间分配某个大小的数据块。然后使用 memcpy() 填充该数据，并将该消息传递给 zmq_msg_send()。
- 要释放（不销毁）消息，则调用 zmq_msg_close()。这会删除一个引用，最终 ØMQ 会销毁该消息。
- 要访问消息内容，可以使用 zmq_msg_data()。要知道消息包含多少数据，可使用 zmq_msg_size()。
- 不要使用 zmq_msg_move()、zmq_msg_copy() 或 zmq_msg_init_data()，除非你已经阅读手册页并精确地知道你为什么需要这些功能。

下面是处理消息的典型代码块，如果你一直在专心阅读，则对它应该是熟悉的。这取自我们在所有的例子中使用的 *zhelpers.h* 文件：

```
// 从套接字接收 ØMQ 字符串并转换成 C 字符串
static char *
s_recv (void *socket) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    int size = zmq_msg_recv (&message, socket, 0);
    if (size == -1)
        return NULL;
    char *string = malloc (size + 1);
    memcpy (string, zmq_msg_data (&message), size);
    zmq_msg_close (&message);
    string [size] = 0;
    return (string);
}

// 将 C 字符串转换成 ØMQ 字符串，并发送到套接字
static int
s_send (void *socket, char *string) {
    zmq_msg_t message;
    zmq_msg_init_size (&message, strlen (string));
    memcpy (zmq_msg_data (&message), string, strlen (string));
    int size = zmq_msg_send (&message, socket, 0);
    zmq_msg_close (&message);
    return (size);
}
```

40

你可以轻松地扩展此代码来发送和接收任意长度的块。



当你向 zmq_msg_send() 传递一个消息时，ØMQ 将清除该消息（即，把它的大小设置为零）。你不能把相同的消息发送两次，并且在发送消息后无法访问它的数据。

如果你想多次发送相同的消息，可以创建第二个消息，并使用 zmq_msg_init() 初始化它，然后用 zmq_msg_copy() 来创建第一个消息的副本。这不复制数据，而是引用它。然后，你就可以将该消息发送两次（或更多，如果你创建多个副本），消息将仅在最后一个副本被发送或关闭时最终被销毁。

ØMQ 还支持多部分（*multipart*）消息，这允许你把帧的列表作为线路上的单个消息发送或接收。这被广泛应用在实际应用程序中，我们将在本章的后面和第 3 章看到它们。

帧（Frame）（在 ØMQ 参考手册页中也称“消息部件”）是 ØMQ 消息的基本线路格式。帧是已指定长度的数据块，此长度可以从零向上。如果做过任何 TCP 编程工作，你就会明白，为什么帧是“现在我应该从该网络套接字读出多少数据？”这个问题的一个有用的答案。



有一个称为 ZMTP 的线路级协议（*http://rfc.zeromq.org/spec:15*），它定义 ØMQ 如何在一个 TCP 连接上读取和写入帧。如果你对其工作原理有兴趣，则可以阅读该规范，它是相当短的。

最初，一个 ØMQ 信息是一帧，像 UDP 一样。后来我们采用多部分消息来扩展了这一点，这是相当简单的带有被设置为 1 的“more”位的帧的序列，接着是一个该位被设置为零的帧。ØMQ API 然后让你写入有一个“more”标志的消息，并且当你读取消息时，它可以让你检查是否存在“more”。

因此，在低级别 ØMQ API 和参考手册中，有关于消息与部件有一些模糊性。因此，下面用一个有用的词汇表来说明：

- 消息可以是一个或多个部件（part）。
- 这些部件也被称为帧（*frames*）。
- 41 每个部件都是一个 zmq_msg_t 对象。
- 你用低级别的 API 分别发送和接收各个部件。
- 高级别 API 为发送整个多部分消息提供包装。

下面是值得了解的一些其他与消息相关的东西：

- 你可以发送长度为零的消息，例如，用于从一个线程把一个信号发送到另一个线程。

- ØMQ 保证提供一个消息所有的部件（一个或多个），或者一个部件也没有。
- ØMQ 不立刻发送消息（单个或多部分），而在以后某些不确定的时间发送。因此，一个多部分消息必须在内存中装入。
- 单部分消息也必须装入内存。如果你想发送任意大小的文件，应该把它们分解成片，并把每一片作为独立的单部分消息发送。
- 在作用域关闭时不自动销毁对象的语言中，在完成消息时，必须调用 `zmq_msg_close()`。

有必要重复的是，仍然不要使用 `zmq_msg_init_data()`，这是一个零拷贝的方法，并保证会给你制造麻烦。

处理多个套接字

迄今为止的大多数例子的主循环如下：

1. 等待套接字上的消息。
2. 处理消息。
3. 重复。

如果我们想在同一时间从多个端点读取，会怎样呢？最简单的方法是将一个套接字连接到所有端点，并让 ØMQ 为我们做扇入。如果远程端点都采用相同的模式，这是合法的，但比如说，如果把一个 PULL 套接字连接到 PUB 端点，则是错误的。

要真正一次性从多个套接字读取，可使用 `zmq_poll()`。一个更好的方法可能是把 `zmq_poll()` 包装到一个框架中，从而把它变成一个很好的事件驱动的反应器（*reactor*），但那明显比我们在这里介绍的要涉及更多的工作。

让我们先从一个不讲究的粗糙例子开始，这样做部分是为了不守规矩的乐趣，但主要是因为它能够让我告诉你如何执行非阻塞套接字读取。示例 2-1 是使用非阻塞读从两个套 42
接字读取的一个简单的例子。这个比较混乱的程序既作为天气状况更新的订阅者，又作为并行任务的工人。

示例2-1：多套接字读取器（msreader.c）

```
//
//  从多个套接字读取
//  此版本使用了一个简单的 recv 循环
//
#include "zhelpers.h"

int main (void)
```

```
{
    //  准备上下文和套接字
    void *context = zmq_ctx_new ();

    //  连接到任务发生器
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  连接到天气服务器
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001", 6);

    //  处理来自两个套接字的消息
    //  对来自任务发生器的流量排优先级
    while (1) {
        //  处理任何等待的任务
        int rc;
        for (rc = 0; !rc; ) {
            zmq_msg_t task;
            zmq_msg_init (&task);
            if ((rc = zmq_msg_recv (&task, receiver, ZMQ_DONTWAIT)) != -1) {
                //  处理任务
            }
            zmq_msg_close (&task);
        }
        //  处理任何等待的天气状况更新
        for (rc = 0; !rc; ) {
            zmq_msg_t update;
            zmq_msg_init (&update);
            if ((rc = zmq_msg_recv (&update, subscriber, ZMQ_DONTWAIT)) != -1) {
                //  处理天气状况更新
            }
            zmq_msg_close (&update);
        }
        //  没有活动，所以休眠 1 毫秒
        s_sleep (1);
    }
    //  我们永远不会到达这里，但无论如何要做一些清理工作
    zmq_close (receiver);
43  zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

这种方法的成本是在第一个消息（当没有消息正在等待处理时，在循环结束时的休眠）

上的一些额外等待时间。对于应用程序来说，如果亚毫秒延迟是至关重要的，这将是一个问题。此外，你还需要检查 nanosleep()（或使用的其他任何函数）的文档，以确保它不会忙于循环。

可以通过从一个套接字读取第一个消息，然后从另一个读取第二个消息这种方式，而不是如我们在本例中对它们排优先级的方式公平地对待套接字。

示例 2-2 显示了正确编写的、相同的、无意义的小应用程序，它使用 zmq_poll()。

示例2-2：多个套接字轮询器（mspoller.c）

```
//
//  从多个套接字读取
//  此版本使用 zmq_poll()
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  连接到任务发生器
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");
    //  连接到天气服务器
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001", 6);

    //  初始化轮询集
    zmq_pollitem_t items [] = {
        { receiver, 0, ZMQ_POLLIN, 0 },
        { subscriber, 0, ZMQ_POLLIN, 0 }
    };
    //  处理来自两个套接字的消息
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmq_msg_init (&message);
            zmq_msg_recv (&message, receiver, 0);
            //  处理任务
            zmq_msg_close (&message);
        }
        if (items [1].revents & ZMQ_POLLIN) {
```

44

```
            zmq_msg_init (&message);
            zmq_msg_recv (&message, subscriber, 0);
            //  处理天气状况更新
            zmq_msg_close (&message);
        }
    }
    //  我们永远不会到达这里
    zmq_close (receiver);
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

items 结构有以下四个成员：

```
typedef struct {
    void *socket;       //  要轮询的 ØMQ 套接字
    int fd;             //  或者，要轮询的本机文件句柄
    short events;       //  要轮询的事件
    short revents;      //  轮询后返回的事件
} zmq_pollitem_t;
```

多部分消息

ØMQ 允许我们撰写由多个帧组成的单个消息，从而给我们一个“多部分消息”。实际的应用程序中相当多地使用了多部分消息，无论是包装带地址信息的消息，还是进行简单的序列化。我们后面将研究应答封包。现在我们将学习如何安全地（但一味地）在需要转发消息但不检查它们的任何应用程序（如代理）中读写多部分消息。

在使用多部分消息时，每个部分都是一个 zmq_msg 条目。例如，如果你要发送的消息具有 5 个部分，你就必须构造、发送，并销毁 5 个 zmq_msg 条目。你既可以事先做到这一点（并把 zmq_msg 条目存储在一个数组或结构中），也可以在你逐一地发送它们时做这项工作。

下面是我们发送多部分消息中的帧的方法（我们把每帧接收到一个消息对象中）：

```
zmq_msg_send (socket, &message, ZMQ_SNDMORE);
...
zmq_msg_send (socket, &message, ZMQ_SNDMORE);
...
zmq_msg_send (socket, &message, 0);
```

下面是我们接收和处理消息中所有部件的方法，无论是单部分还是多部分消息，都是如此：

```
while (1) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_msg_recv (socket, &message, 0);
    //  处理消息帧
    zmq_msg_close (&message);
    int more;
    size_t more_size = sizeof (more);
    zmq_getsockopt (socket, ZMQ_RCVMORE, &more, &more_size);
    if (!more)
        break;      //  最后一个消息帧
}
```


关于多部分消息，有些事情需要明白：

- 当发送一个多部分消息时，仅当你发送最后的部分时，第 1 部分和以下的所有部分才会在线路上实际发送。
- 如果你使用的是 zmq_poll()，当你收到一条消息的第一部分时，其余部分也都已经到达了。
- 你要么会收到一条消息的所有部分，要么根本连一个部分也不会收到。
- 一个消息的每个部分都是一个单独的 zmq_msg 条目。
- 无论你是否设置 RCVMORE 选项，都将收到一条消息的所有部分。
- 在发送时，ØMQ 消息帧都在内存中排队，直到最后一个消息帧被接收为止，然后再一起发送它们。
- 除非关闭套接字，否则没有办法取消部分发送的消息。

中间层及代理

ØMQ 旨在分散智能，但是，这并不意味着你的网络在中间是空白的。相反，它充满了信息感知的基础设施，而且经常，你用 ØMQ 来构建基础设施。ØMQ 管道的范围可以包含从微小的管道直到完全成熟的面向服务的代理。消息传递业界将此称为中间层（*intermediation*），这意味着在中间处理两边的东西。在 ØMQ 中，我们称之为代理（proxy）、队列、转发器（forwarder）、设备或中间人（broker），视上下文而定。

在现实世界中，这种模式是非常普遍的，这就是为什么我们的社会和经济都充满了中间人的原因，除了降低较大的网络的复杂性和扩展成本，它们没有其他真正的功能。现实世界中的中间人通常被称为批发商、分销商、经理，等等。

动态发现问题

在设计大型的分布式体系结构时，你会遇到的问题之一是——发现。也就是说，部件如何认识对方呢？如果部件来来去去，这是特别困难的，因此，我们可以称之为“动态发现问题”。

46 实现动态发现有几种解决方案。最简单的是通过硬编码（或配置）的网络架构来完全避免它，从而通过手工完成发现。也就是说，每当你添加一个新的部件，你就要重新配置网络来认识它。

在实践中，这种方案导致了结构变得越来越脆弱和笨重。比方说，你有一个发布者和一百个订阅者。你可以通过在每个订阅者处配置发布者端点来将每个订阅者连接到发布服务器。这很简单（参见图 2-4），订阅者是动态的，而发布者是静态的。现在，假设你添加更多的发布者。突然，这变得不再那么容易了。如果继续将每个订阅者都连接到每个发布者，这种避免动态发现的方法的成本就会变得越来越高。

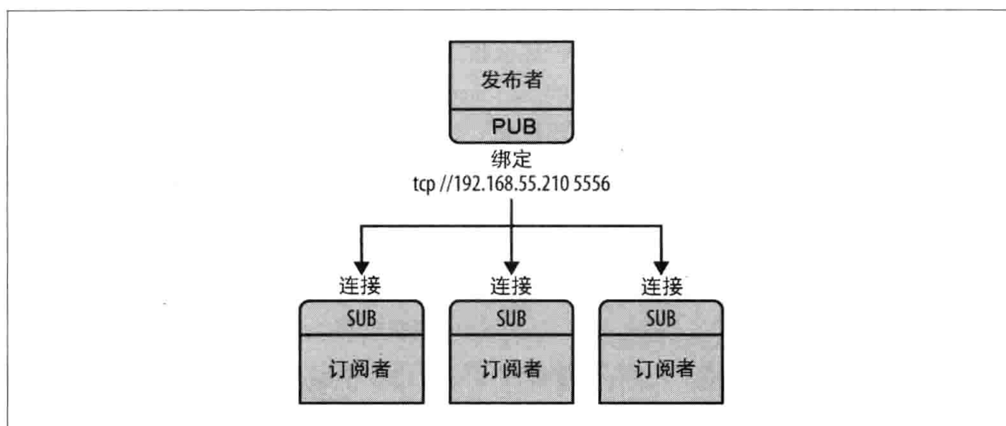


图2-4：小规模的发布-订阅网络

对于这一问题，有相当多的解决方案，但最简单的方法就是增加一个中间层，即，网络中一个静态的点，而让所有其他节点都连接到它。在经典的消息传递中，这是消息代理的工作。ØMQ 不提供这样的消息代理，但它可以让我们很容易地建立中间层。

你可能想知道，如果所有的网络最终变得足够大，从而需要中间层，为什么我们不简单地为所有的应用程序在合适的地方设置一个消息代理呢？对于初学者来说，这是一个公平的妥协。只是始终使用星形拓扑结构，忘掉性能，事情通常会奏效。然而，消息代理是贪婪的东西，在它们作为中心的中间层起作用时，它们变得太复杂，太有状态，并最

终出现问题。

最好将中间层当作简单的无状态的消息交换机。最好的比喻是一个 HTTP 代理服务器，它存在，但并没有任何特殊的作用。在我们的例子中，添加发布 - 订阅代理解决了动态发现问题。我们在网络的“中间”设置代理（参见图 2-5）。代理打开一个 XSUB 套接字和 XPUB 套接字，并将它们绑定到各知名 IP 地址和端口。然后，所有其他的进程都连接到代理，而不是互相连接。这样一来，添加更多用户或发布者就变得微不足道了。 47

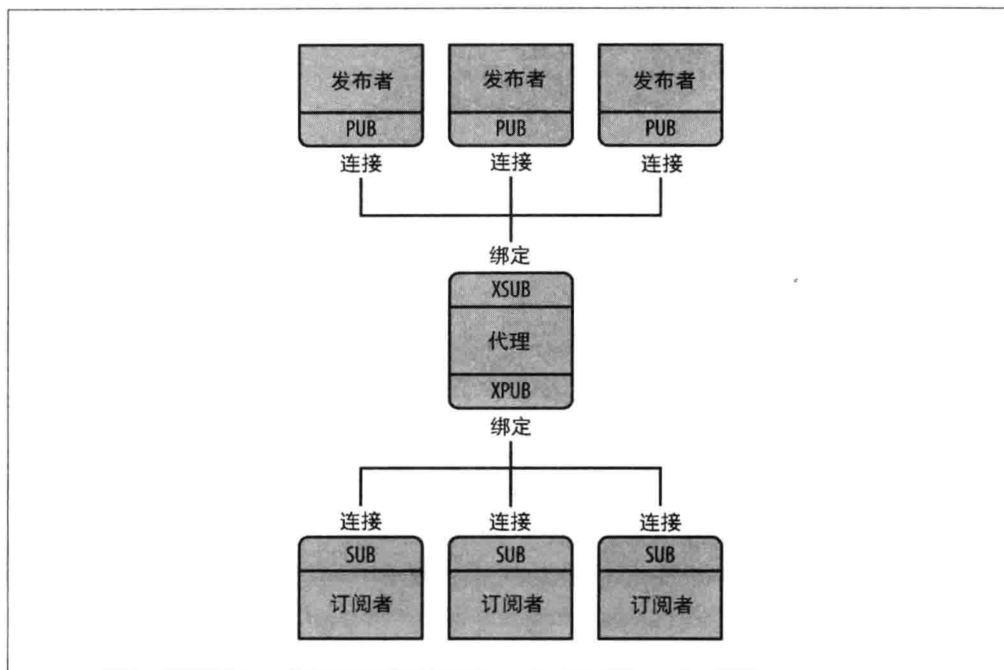


图2-5：使用代理的发布-订阅网络

我们需要 XPUB 和 XSUB 套接字，因为 ØMQ 执行订阅转发：SUB 套接字实际上将订阅作为特殊消息发送到 PUB 套接字。代理也必须转发这些，这通过从 XPUB 套接字读取它们，并将它们写入到 XSUB 套接字来实现。这是 XSUB 和 XPUB 的主要用例（参见图 2-6）。

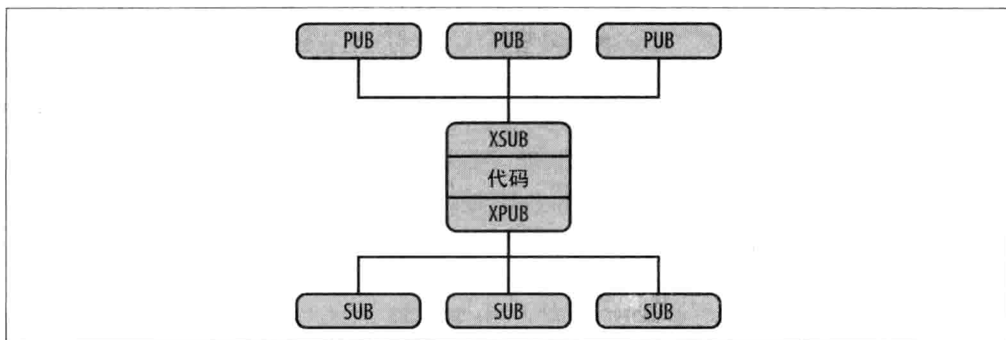


图2-6：扩展的发布-订阅

48 共享队列（DEALER 和 ROUTER 套接字）

在“Hello World”客户端/服务器应用程序中，我们有一个客户端，它与一个服务交流。然而，在实际情况下，我们通常需要允许多个服务，以及多个客户端相互交流。这让我们扩大了服务的威力（多线程、进程或节点，而不是只有一个）。唯一的限制是，这些服务必须是无状态的，所有的状态都附带在请求中或在某些共享存储中，例如在一个数据库中。

将多个客户端连接到多个服务器有两种方法。蛮力的方法是将每个客户端套接字连接到多个服务端点。一个客户端套接字可以连接到多个服务套接字，而 REQ 套接字随后会将请求分配给这些服务。比如说，你将客户端套接字连接到三个服务端点：A、B 和 C。客户端发出请求 R1、R2、R3、R4。R1 和 R4 连到服务 A，R2 连到 B，而 R3 连到服务 C（参见图 2-7）。

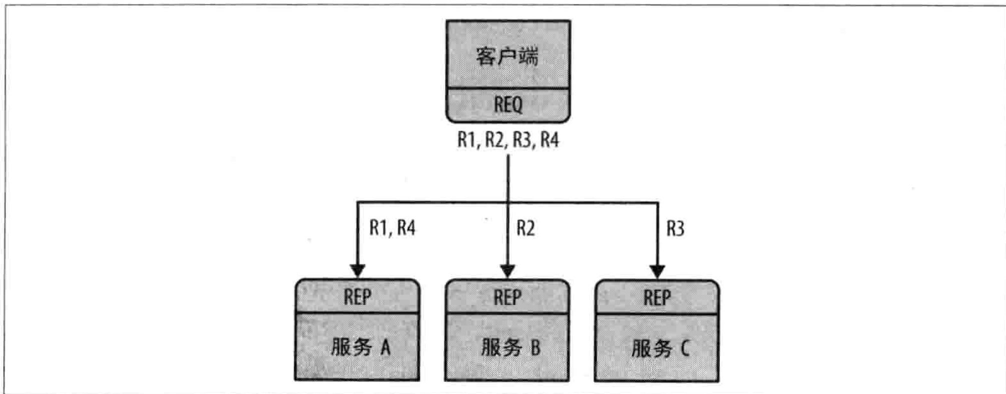


图2-7：请求分配

这种设计可以让你廉价地添加更多的客户端，你还可以添加更多的服务。每个客户端将其分配到请求的服务，但每个客户端必须知道服务的拓扑结构。如果你有 100 个客户端，然后你决定要额外添加 3 个服务，则需要重新配置并重新启动 100 个客户端，以便所有客户端都能识别这 3 个新服务。

当我们的超级计算集群耗尽了资源的时候，我们就迫切需要增加几百个新的服务节点，这显然不是我们想要在凌晨 3 点做的工作。太多的静态部件就像液态混凝土：知识是分布式的，而你拥有的静态部件越多，改变拓扑结构花的精力就越多。我们想要的是位于客户端服务之间，集中所有拓扑知识的东西。理想情况下，我们应该能够在任何时间添加和移除服务或客户端，而不触动拓扑结构的其他部分。

因此，我们将编写一个小型消息排队代理，以使我们具备这种灵活性。该代理绑定到两个端点，一个用于客户端的前端，另一个用于服务的后端。然后，它使用 zmq_poll() 来监控这两个套接字的活动，而当它有一些活动时，就会在它的两个套接字之间频繁往返消息。它实际上并不显式管理任何队列——ØMQ 会自动在每个套接字上管理它们。 49

当你使用 REQ 与 REP 交流时，你会得到一个严格同步的请求 - 应答对话。客户端发送一个请求，该服务读取请求并发送应答。然后，客户端读取应答。如果客户端或服务尝试做别的事（例如，连续发送两个请求，而无须等待响应），那么它会得到一个错误。

但是，我们的代理必须是非阻塞的。很显然，可以使用 zmq_poll() 来等待任何一个套接字上的活动，但我们不能用 REP 和 REQ。

幸运的是，有两个称为 DEALER（经销商）和 ROUTER（路由器）的套接字，它们让你能够执行无阻塞的请求 - 响应。在第 3 章中，你将看到如何用 DEALER 和 ROUTER 套接字来构建各种异步请求 - 应答流。现在，我们只是去看 DEALER 和 ROUTER 如何让我们跨越中间层，也就是跨越我们的小型代理来扩展 REQ-REP。

在这个简单的扩展请求 - 应答模式中，REQ 与 ROUTER 交流，而 DEALER 与 REP 交流。在 DEALER 和 ROUTER 之间，我们用代码（如我们的代理）从一个套接字提取消息，并将消息推送到其他的套接字（参见图 2-8）。

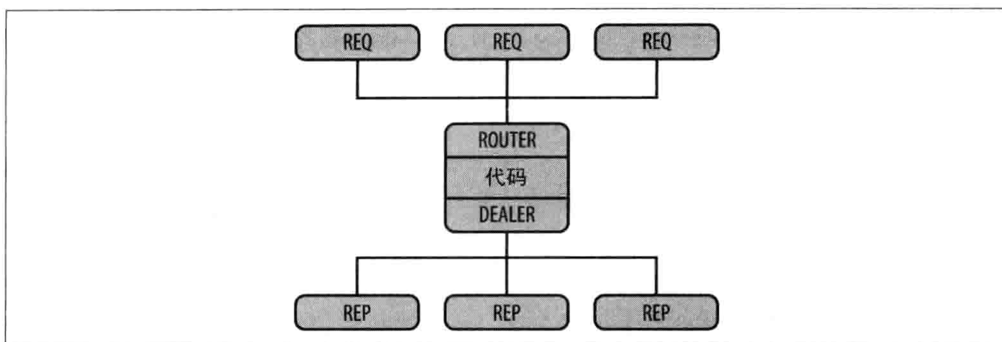


图2-8：扩展的请求-应答

50 请求 - 应答代理绑定到两个端点，一个用于要连接到的客户端（前端套接字），另一个用于连接工人（后端）。为了测试这个代理，你会希望改变你的工人，以将它们连接到后端套接字。示例 2-3 是一个客户端，它显示了我的意图。

示例2-3：请求-应答客户端（rrclient.c）

```
//
//  Hello World 客户端
//  连接 REQ 套接字到 tcp://localhost:5559
//  发送“Hello”到服务器，期待它返回“World”
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  用来与服务器交流的套接字
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5559");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        s_send (requester, "Hello");
        char *string = s_recv (requester);
        printf ("Received reply %d [%s]\n", request_nbr, string);
        free (string);
    }
    zmq_close (requester);
    zmq_ctx_destroy (context);
    return 0;
}
```

工人的代码如示例 2-4 所示。

示例2-4：请求-应答工人（rrworker.c）

```
//
//  Hello World 工人
//  连接 REP 套接字到 tcp://*:5560
//  期待从客户端发出的“Hello”，用“World”来应答
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  用来与客户端交流的套接字
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_connect (responder, "tcp://localhost:5560");

    while (1) {                                                    51
        //  等待来自客户端的下一个请求
        char *string = s_recv (responder);
        printf ("Received request: [%s]\n", string);
        free (string);

        //  执行一些“工作”
        sleep (1);

        //  将应答发回到客户端
        s_send (responder, "World");
    }
    //  我们永远不会到达这里，但如果由于某种原因到达了，就执行清理工作
    zmq_close (responder);
    zmq_ctx_destroy (context);
    return 0;
}
```

可以正确地处理多部分消息的代理的最终源代码如示例 2-5 所示。

示例2-5：请求-应答代理（rrbroker.c）

```
//
//  简单请求 - 应答代理
//
#include "zhelpers.h"

int main (void)
```

```
{
    //  准备上下文和套接字
    void *context = zmq_ctx_new ();
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (frontend, "tcp://*:5559");
    zmq_bind (backend,  "tcp://*:5560");

    //  初始化轮询集合
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    //  在套接字之间切换消息
    while (1) {
        zmq_msg_t message;
        int more;               //  多部分检测

        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            while (1) {
                //  处理消息的所有部分
52              zmq_msg_init (&message);
                zmq_msg_recv (&message, frontend, 0);
                size_t more_size = sizeof (more);
                zmq_getsockopt (frontend, ZMQ_RCVMORE, &more, &more_size);
                zmq_msg_send (&message, backend, more? ZMQ_SNDMORE: 0);
                zmq_msg_close (&message);
                if (!more)
                    break;      //  消息的最后部分
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            while (1) {
                //  处理消息的所有部分
                zmq_msg_init (&message);
                zmq_msg_recv (&message, backend, 0);
                size_t more_size = sizeof (more);
                zmq_getsockopt (backend, ZMQ_RCVMORE, &more, &more_size);
                zmq_msg_send (&message, frontend, more? ZMQ_SNDMORE: 0);
                zmq_msg_close (&message);
                if (!more)
                    break;      //  消息的最后部分
            }
        }
```

```
    }
    //  我们永远不会到达这里，但如果由于某种原因到达了，就执行清理工作
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

使用一个请求 - 应答代理让你的客户端 / 服务器架构易于扩展，因为客户端看不到工人而工人也看不到客户端。在中间的代理是唯一的静态节点（参见图 2-9）。

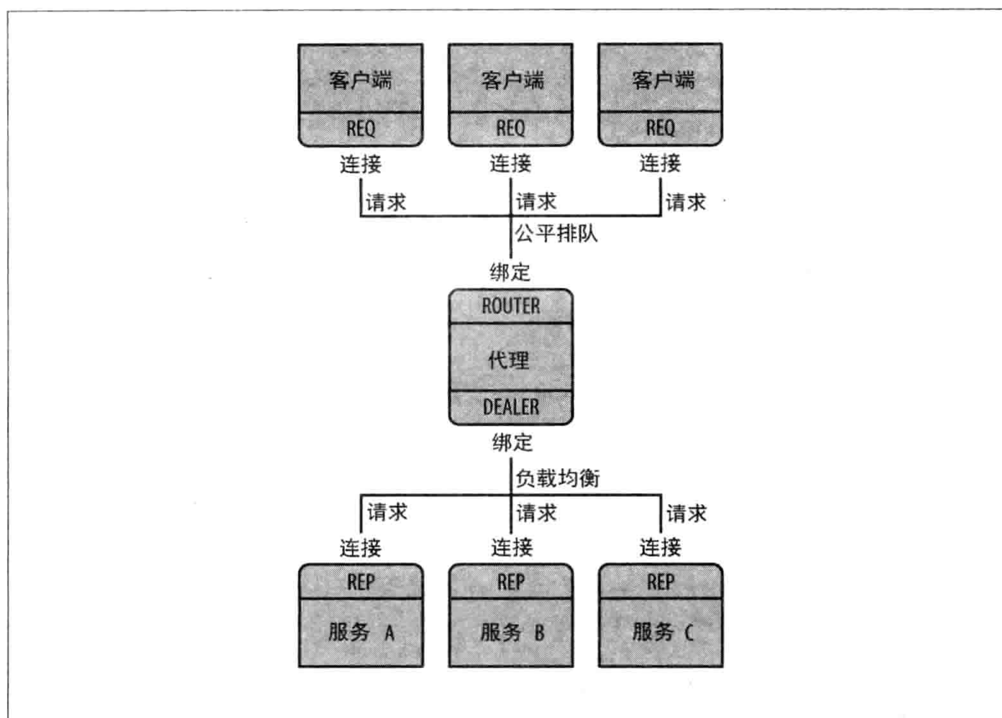


图2-9：请求-应答代理

ØMQ 的内置代理功能

53

事实证明，上一节的 *rrbroker* 核心循环是非常有用的，并且可重复使用。它可以让我们用很少的努力建立发布 - 订阅代理和共享队列，以及其他的小中间层。ØMQ 将这个封装在单个方法中，即 zmq_proxy() 中：

```
zmq_proxy (frontend, backend, capture);
```

两个套接字（或三个，如果我们想要捕获数据）必须被正确地连接、绑定和配置。当我们调用 zmq_proxy() 方法时，它酷似开始 *rrbroker* 的主循环。让我们重写请求 - 应答代理来调用 zmq_proxy()，并把它重新命名为一个昂贵的冠冕堂皇的“消息队列”（人们已经为做很少工作的代码做了很多管理工作）。示例 2-6 显示了实现的结果。

示例2-6：消息队列代理（msgqueue.c）

```
//
//  简单消息队列代理
//  与请求 - 应答代理相同，但使用 QUEUE 设备
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  面对客户端的套接字
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    int rc = zmq_bind (frontend, "tcp://*:5559");
    assert (rc == 0);

    //  面对服务的套接字
    void *backend = zmq_socket (context, ZMQ_DEALER);
    rc = zmq_bind (backend, "tcp://*:5560");
    assert (rc == 0);

    //  启动代理
    zmq_proxy (frontend, backend, NULL);

    //  我们永远不会到达这里……
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

54

如果你像大多数 ØMQ 用户一样，那么在这个阶段你会开始想，“如果我为代理插上随机套接字类型，我可以做什么样的邪恶东西呢？”对此问题简短的回答是：尝试并弄明白正在发生的事情。在实践中，你通常会坚持用 ROUTER/DEALER、XSUB/XPUB 或 PULL/PUSH。

传输桥接

ØMQ 用户的一个常见要求是，“我如何用 X 技术连接我的 ØMQ 网络？”其中 X 是某些其他网络或通信技术。简单的答案是建立一个“桥梁”。桥是指一个小型应用程序，它用一种协议与一个套接字交流，并将其转换为另一个套接字的另一种协议。如果你喜欢，可以称之为一个协议解释器。ØMQ 中一个常见的桥接问题是衔接两种传输协议或网络。

举一个例子，假设我们要编写一个小型代理（参见示例 2-7），它位于一个发布者和一组订阅者之间，衔接两个网络。前端接口（SUB）面对着驻留了天气服务器的内部网络，而后端（PUB）面对着外部网络上的订阅者。它在前端套接字上订阅气象服务，并在后端套接字上重新发布数据（参见图 2-10）。

示例2-7：天气更新代理（wuproxy.c）

```
//
//  天气代理设备
//
#include "zhelpers.h"

int main (void)
{                                                                    55
    void *context = zmq_ctx_new ();

    //  天气服务器所在的位置
    void *frontend = zmq_socket (context, ZMQ_XSUB);
    zmq_connect (frontend, "tcp://192.168.55.210:5556");

    //  用于订阅者的公共端点
    void *backend = zmq_socket (context, ZMQ_XPUB);
    zmq_bind (backend, "tcp://10.1.1.0:8100");

    //  持续运行代理直到用户中断它为止
    zmq_proxy (frontend, backend, NULL);

    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

它看起来非常类似于早期代理的例子，但关键的部分在于，前端和后端套接字在两个 56
不同的网络上。例如，我们可以利用这个模型将多播网络（pgm 传输协议）连接到一个 TCP 发布者。

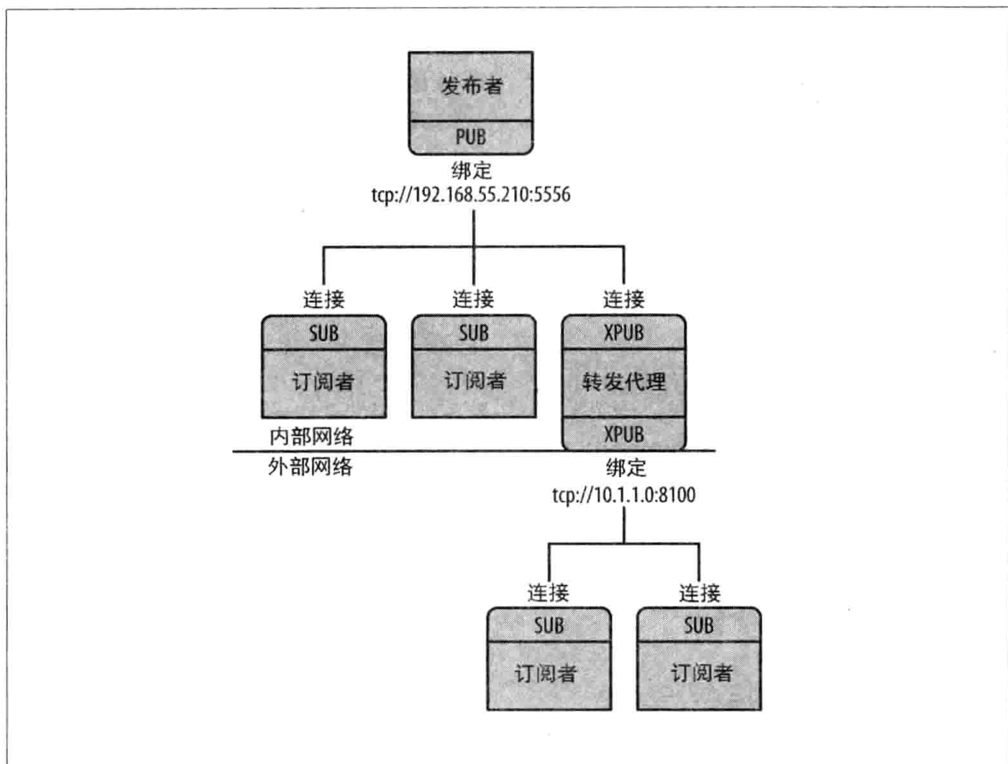


图2-10：发布-订阅转发代理

处理错误和 ETERM

ØMQ 的错误处理哲学是快速失败和弹性的组合。我们认为，进程应禁不起内部错误的任何微小伤害，但同时它应尽可能强劲地抵抗外部攻击和错误。打一个比方，如果一个活细胞检测到一个内部错误，它就会自我毁灭，但它会通过一切可能的手段抵御外部的攻击。

断言对于健壮的代码绝对是至关重要的，这刺激了 ØMQ 代码，它们只需要在细胞壁正确的一侧，并且应该有这样的壁。如果不清楚故障是内部的还是外部的，这就证明存在一个需要被修复的设计缺陷。在 C/C++ 中，断言会导致该应用程序立即停止并报错。在其他语言中，你可能会得到异常或暂停。

当 ØMQ 检测到外部故障时，它会将一个错误返回给调用代码。在某些罕见的情况下，如果没有从错误中恢复的明显策略，它就会默默地丢弃消息。

到目前为止，大多数我们看到的 C 实例都没有错误处理。实际的代码应该对每一个 ØMQ 调用执行错误处理。如果你使用的绑定语言不是 C，绑定可能会为你处理错误。但在 C 语言中，你需要自己做这个工作。这里有一些简单的规则，首先是 POSIX 的约定：

- 如果创建对象的方法失败，它们就返回 NULL。
- 处理数据的方法可以返回处理的字节数，如果发生错误或故障，返回 -1。
- 其他方法返回 0 表示成功，返回 -1 表示错误或失败。
- 错误代码在 errno 或 zmq_errno() 中提供。
- 描述错误的文字记录是由 zmq_strerror() 提供的。

例如：

```
void *context = zmq_ctx_new ();
assert (context);
void *socket = zmq_socket (context, ZMQ_REP);
assert (socket);
int rc = zmq_bind (socket, "tcp://*:5555");
if (rc != 0) {
    printf ("E: bind failed: %s\n", strerror (errno));
    return -1;
}
```

你可能要处理的非致命性的例外情况主要有两种： 57

- 当一个线程使用 ZMQ_DONTWAIT 选项调用 zmq_msg_recv() 并且没有等待数据时，ØMQ 将返回 -1，并将 errno 设置为 EAGAIN。
- 当一个线程调用 zmq_ctx_destroy() 而其他线程正在做阻塞工作时，zmq_ctx_destroy() 调用关闭该上下文，所有的阻塞调用都用 -1 退出，并将 errno 设置为 ETERM。

在 C/C++ 中，会在优化的代码中将断言完全删除，所以不要犯将整个 ØMQ 调用包装在一个 assert() 中的错误。它看起来整洁，不过随后优化器会移除所有的断言和你想执行的调用，而应用程序会以令人印象深刻的方式中断。

让我们来看看如何彻底地关闭进程。我们将利用上一节中的并行流水线的例子。如果我们已经在后台启动了一大堆工人，而现在想要在批处理完成后清除他们，那么让我们通过发送一个 kill 消息给工人来做到这一点。完成这个工作的最好的地方是接收器，因为它确实知道该批处理在什么时候完成。

我们如何将接收器连接到工人呢？ PUSH/PULL 套接字只是单向的。标准的 ØMQ 解决办法是：为你需要解决的每个类型的问题创建一个新的套接字流。我们将使用一个发布

-订阅模型来将 kill 消息发送给工人：

- 接收器在新的端点上创建一个 PUB 套接字。
- 工人将自己的输入套接字绑定到这个端点。
- 当接收器检测到批处理结束时，它会将一个 kill 消息发送给其 PUB 套接字。
- 当一个工人检测到这个 kill 消息时，它就会退出。

接收器并不需要很多新代码：

```
void *control = zmq_socket (context, ZMQ_PUB);
zmq_bind (control, "tcp://*:5559");
...
//  发送 kill 信号给工人
zmq_msg_init_data (&message, "KILL", 5);
zmq_msg_send (control, &message, 0);
zmq_msg_close (&message);
```

图 2-11 显示了生成的配置。

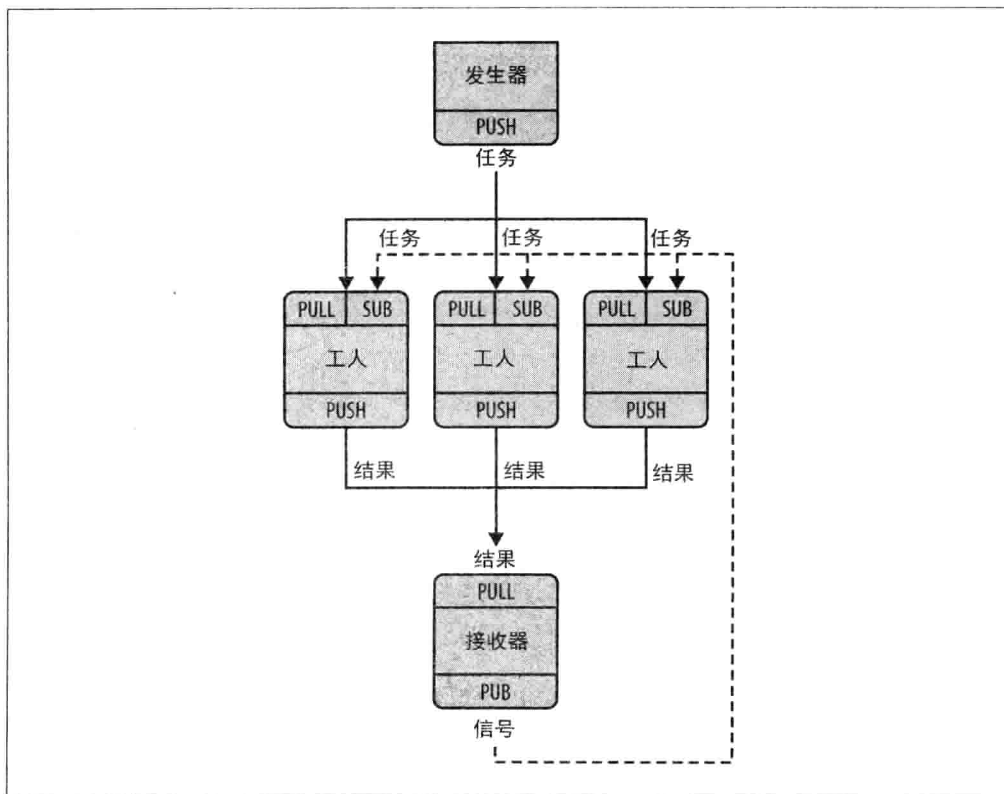


图2-11：使用kill信号的并行流水线

示例 2-8 中包含了工作进程的代码，它使用我们在前面看到的 zmq_poll() 方法，它管理两个套接字（一个 PULL 套接字获取任务，一个 SUB 套接字获取控制指令）。 58

示例2-8：使用kill信号的并行任务工人（taskwork2.c）

```
//
//  任务工人 – 设计 2
//  添加发布 - 订阅流来接收和响应 kill 信号
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  用于接收消息的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  消息发往的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sender, "tcp://localhost:5558");

    //  用于控制输入的套接字
    void *controller = zmq_socket (context, ZMQ_SUB);
    zmq_connect (controller, "tcp://localhost:5559");
    zmq_setsockopt (controller, ZMQ_SUBSCRIBE, "", 0);

    //  处理来自接收者和控制器的消息
    zmq_pollitem_t items [] = {
        { receiver, 0, ZMQ_POLLIN, 0 },
        { controller, 0, ZMQ_POLLIN, 0 }
    };
    //  处理来自两个套接字的消息
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmq_msg_init (&message);
            zmq_msg_recv (&message, receiver, 0);

            //  执行工作
            s_sleep (atoi ((char *) zmq_msg_data (&message)));

            //  将结果发送到接收器
            zmq_msg_init (&message);
```

59

```
            zmq_msg_send (&message, sender, 0);

            // 用于查看器的简易过程指示器
            printf (".");
            fflush (stdout);

            zmq_msg_close (&message);
        }
        // 任何等待中的控制器命令相当于“KILL”
        if (items [1].revents & ZMQ_POLLIN)
            break;                          // 退出循环
    }
    // 完成
    zmq_close (receiver);
    zmq_close (sender);
    zmq_close (controller);
    zmq_ctx_destroy (context);
    return 0;
}
```

示例 2-9 显示了修改后的接收器应用程序。当它完成对结果的收集时，它广播 kill 消息给所有的工人。

60 示例2-9：使用kill信号的并行任务接收器（tasksink2.c）

```
//
// 任务接收器 — 设计 2
// 添加发布 - 订阅流来将 kill 信号发送给工人
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    // 用于接收消息的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    // 用于工人控制的套接字
    void *controller = zmq_socket (context, ZMQ_PUB);
    zmq_bind (controller, "tcp://*:5559");

    // 等待批处理开始
    char *string = s_recv (receiver);
    free (string);
```

```
    // 启动时钟
    int64_t start_time = s_clock ();

    // 处理100个确认
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if ((task_nbr / 10) * 10 == task_nbr)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    printf ("Total elapsed time: %d msec\n",
        (int) (s_clock () - start_time));

    // 发送kill信号给工人
    s_send (controller, "KILL");

    // 完成
    sleep (1);              // 给ØMQ时间来传递

    zmq_close (receiver);
    zmq_close (controller);
    zmq_ctx_destroy (context);
    return 0;
}
```

处理中断信号

现实中的应用程序需要使用 Ctrl-C 或其他信号，如 SIGTERM 中断来彻底地关闭。默认情况下，它们简单地杀掉进程，这会造成消息不会被刷新、文件将无法完全关闭等一系列问题。

示例 2-10 显示了我们如何用各种语言处理信号。

示例2-10：彻底地处理Ctrl-C（interrupt.c）

```
//
// 显示如何处理Ctrl-C
//
#include <zmq.h>
#include <stdio.h>
```

```
#include <signal.h>

//  ---------------------------------------------------------------------
//  信号处理
//
//  在应用程序启动时调用 s_catch_signals()，并且如果 s_interrupted 是 1，
//  则退出主循环。特别适合与 zmq_poll 一起使用。

static int s_interrupted = 0;
static void s_signal_handler (int signal_value)
{
    s_interrupted = 1;
}

static void s_catch_signals (void)
{
    struct sigaction action;
    action.sa_handler = s_signal_handler;
    action.sa_flags = 0;
    sigemptyset (&action.sa_mask);
    sigaction (SIGINT, &action, NULL);
    sigaction (SIGTERM, &action, NULL);
}

int main (void)
{
    void *context = zmq_ctx_new ();
    void *socket = zmq_socket (context, ZMQ_REP);
    zmq_bind (socket, "tcp://*:5555");

    s_catch_signals ();
    while (1) {
        //  阻塞读会在收到一个信号时退出
        zmq_msg_t message;
        zmq_msg_init (&message);
        zmq_msg_recv (&message, socket, 0);

62      if (s_interrupted) {
            printf ("W: interrupt received, killing server...\n");
            break;
        }
    }
    zmq_close (socket);
    zmq_ctx_destroy (context);
    return 0;
}
```

该程序提供了 s_catch_signals()，它捕获 Ctrl-C（SIGINT）和 SIGTERM。当这些信号中的某个到达时，s_catch_signals() 处理程序会设置全局变量 s_interrupted。感谢你的信号处理程序，你的应用程序不会自动死掉。相反，你有机会执行清理并正常退出。你现在要显式地检查中断并妥善处理它。在你的主代码开始处，通过调用 s_catch_signals()（从 *interrupt.c* 复制这个代码）来执行此操作。此操作设置了信号处理。中断将在如下方面影响 ØMQ 调用：

- 如果你的代码在 zmq_msg_recv()、zmq_poll() 或 zmq_msg_send() 中阻塞，当信号到达时，调用将返回 EINTR。
- 如果类似 s_recv() 的包装器被中断，它们就返回 NULL。

因此，检查是否有 EINTR 返回码、NULL 返回值或 s_interrupted。

下面是一个典型的代码片段：

```
s_catch_signals ();
client = zmq_socket (...);
while (!s_interrupted) {
    char *message = s_recv (client);
    if (!message)
        break;          //  使用了 Ctrl-C
}
zmq_close (client);
```

如果调用 s_catch_signals()，并且不对中断进行测试，你的应用程序就会对 Ctrl-C 和 SIGTERM 免疫，这可能是有用的，但通常并非如此。

检测内存泄漏

任何长期运行的应用程序都必须正确地管理内存，否则它最终会耗尽所有可用内存并崩溃。如果你使用的是自动为你管理内存的语言，那么祝贺你。如果你用 C 或 C++ 或任何其他需要你负责内存管理的语言编写程序，那么这里有一个使用 *valgrind*（*http://valgrid.org*）的简短的教程，这个工具将对你的程序的任何内存泄漏出具报告：

- 要在 Ubuntu 或 Debian 操作系统上安装 valgrind，请输入： 63
  ```
  sudo apt-get install valgrind
  ```
- 默认情况下，ØMQ 会导致 valgrind 输出很多错误信息。要删除这些警告信息，请创建一个名为 *valgrind.supp* 的文件，其中包含如下的内容：

```
{
   <socketcall_sendto>
   Memcheck:Param
   socketcall.sendto(msg)
   fun:send
   ...
}
{
   <socketcall_sendto>
   Memcheck:Param
   socketcall.send(msg)
   fun:send
   ...
}
```

- 修复你的应用程序，使得它在按Ctrl-C后完全退出。对于本身会退出的任何应用程序，这是没有必要的，但对于长期运行的应用程序，这是必不可少的。否则 valgrind 会输出目前所有已分配内存的信息。
- 用 *-DDEBUG* 构建你的应用程序，如果它不是你的默认设置。这可以确保 valgrind 可以告诉你内存究竟在何处泄漏。
- 最后，用如下命令运行 valgrind（全部在一行上输入）

  ```
  valgrind --tool=memcheck --leak-check=full --suppressions=valgrind.supp
  someprog
  ```

修复它报告的全部错误之后，你应该得到令人愉快的消息：

```
==30536== ERROR SUMMARY: 0 errors from 0 contexts...
```

使用 ØMQ 编写多线程程序

ØMQ 也许是有史以来编写多线程（MT）应用程序最好的方法。如果你习惯使用传统的套接字，那么使用 ØMQ 套接字需要做一些调整，ØMQ 多线程编程将把你了解到的编写 MT 应用程序的一切东西抛到花园中的一个土堆里，用汽油浇在它上面，并点火烧掉。一本书值得烧掉，这是罕见的，但大多数并发编程的书籍确实应该被烧掉。

为了编写完美的 MT 程序（我的意思是，从根本上），除跨 ØMQ 套接字发送的消息外，我们不需要互斥量、锁，或任何其他形式的线程间的通信。

64 我所说的“完美”的 MT 程序，是指代码很容易编写和理解，适用任何编程语言和任何操作系统相同的设计方法，并且可以在零等待状态按任意数量的 CPU 扩展，而且没有收益递减点的程序。

如果你已经花了几年时间学习各种技巧，使用锁、信号量和临界区，才让你的 MT 代码能够完全正常地工作，更不用说能够迅速运行了，那么当你意识到这一切都无济于事时，你会非常反感。如果我们已经从 30 年以上的并发编程中学到了一个教训，那它就是：只要不共享状态。这就像两个酒鬼试图共享一瓶啤酒，这与他们是否是好哥们无关。迟早，他们会打起来。而你添加到表中的“酒鬼”越多，他们互相为抢啤酒喝就打得越凶。大多数 MT 应用程序的惨象就像喝醉的酒鬼在酒吧打架。

在你编写经典的共享状态的 MT 代码时，因为似乎能正常工作的代码会在压力下突然失败，所以，如果你需要与之斗争的怪问题的清单没有直接转化为压力和风险，那么这将是值得庆幸的。几年前，一个拥有世界一流的代码排错经验的大公司发布了“在你的多线程代码中可能出现的 11 大问题”，其中包括被遗忘的同步、不正确的粒度、读写冲突、无锁重新排序、锁护送效应、两步过程和优先级倒置。

是啊，我们归纳出了 7 大问题，而不是 11 大问题。但这不是问题的关键，问题的关键是，你真的要在繁忙的周四下午 3 点，让在电网或股市系统运行的代码开始获取两步锁护送吗？谁在乎这些术语实际上是什么意思呢？与越来越复杂的副作用以及比以往任何时候都更复杂的黑客攻击做斗争，这不是激励我们去编程的东西。

一些广泛使用的模型，尽管是依据整个行业做出的，但从根本上来说是坏的模型，共享状态并发性就是其中之一。希望无限扩展的代码通过发送消息和不共享除了对状态不佳的编程模型的蔑视以外的东西，就能够像互联网一样做到无限扩展。

若要使用 ØMQ 快乐地编写多线程的代码，你应该遵循如下这些规则：

- 你不能从多个线程访问相同的数据。使用传统的 MT 技术，如互斥，在 ØMQ 应用程序中是一种反模式。这里唯一的例外是 ØMQ 上下文对象，它是线程安全的。
- 你必须为你的进程创建一个 ØMQ 上下文，并将它传递给你想要通过 `inproc` 套接字来连接的所有线程。
- 你可以把线程作为拥有自己的上下文的独立任务，但这些线程不能通过 `inproc` 通信。不过，以后它们会更容易分解成独立的进程。
- 你不能在线程之间共享 ØMQ 套接字。ØMQ 套接字不是线程安全的。从技术上讲，做到这一点是可能的，但它要求信号量、锁或互斥，这将使得你的应用程序缓慢且脆弱。线程之间共享套接字唯一完全合乎情理的地方，是语言绑定，它需要在套接 65
字上做类似垃圾收集的神奇操作。

例如，如果你需要在应用程序中启动多个代理，你会想让每个代理在自己的线程中运行。在一个线程中创建代理前端和后端套接字，然后将套接字传递给在另一个线程中的代理，这是很容易出错的。这似乎能够工作，但将会随机失败。记住：不要在创建套接字的线

程外使用或关闭该套接字。

如果你遵循这些规则，当有需要时，你就可以很容易地将线程分割成独立的进程。应用逻辑可以位于线程、进程或节点上：无论你的规模需求是什么。

ØMQ 使用本机操作系统线程，而不是虚拟的“绿色”线程。其优点是你不需要学习任何新的线程 API，并且 ØMQ 线程清楚地映射到操作系统。可以使用如 Intel 的 ThreadChecker 标准工具来查看你的应用程序在做什么。缺点是，在有的时候，例如在启动新的线程时，你的代码将不能移植，而且，如果你的线程数量庞大（数以千计），那么某些操作系统会被压垮。

让我们来看看它实际上是如何工作的。我们将会把我们的旧 Hello World 服务器变成更有能力的东西。原来的服务器运行在一个单独的线程中。如果每个请求的工作量很少，这很好：一个 ØMQ 线程可以全速在一个 CPU 内核上运行，没有等待，并完成非常多的工作。但现实的服务器必须针对每个请求做不平凡的工作。当 10000 个客户端同时点击服务器时，单个核心可能就不够了。因此，实际的服务器必须启动多个工作线程。然后，它尽量快速地接受请求，并将这些请求分配给它的工作线程。工作线程通过艰难的工作，并最终发回它们的应答。

当然，你也可以使用代理和外部的工作进程做这一切工作，但启动一个吃掉 16 个内核的进程比启动 16 个每个吞噬了一个核心的进程往往更容易。此外，以线程方式运行工人将产生一个网络跃点、延迟和网络流量。

示例 2-11 中的 Hello World 服务的 MT 版本基本上是将代理和工人折叠到一个单独的进程中。我们使用 pthreads，因为它是多线程编程的最普遍的标准。

示例2-11：多线程服务（mtserver.c）

```
//
//  多线程 Hello World 服务器
//
#include "zhelpers.h"
#include <pthread.h>
```

66

```
static void *
worker_routine (void *context) {
    //  与调度器交流的套接字
    void *receiver = zmq_socket (context, ZMQ_REP);
    zmq_connect (receiver, "inproc://workers");

    while (1) {
        char *string = s_recv (receiver);
```

```
        printf ("Received request: [%s]\n", string);
        free (string);
        //  执行某项“工作”
        sleep (1);
        //  将应答发回客户端
        s_send (receiver, "World");
    }
    zmq_close (receiver);
    return NULL;
}

int main (void)
{
    void *context = zmq_ctx_new ();

    //  与客户端交流的套接字
    void *clients = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (clients, "tcp://*:5555");

    //  与工人交流的套接字
    void *workers = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (workers, "inproc://workers");

    //  工人线程的投放池
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_routine, context);
    }
    //  通过一个排队代理将工作线程连接到客户端线程
    zmq_proxy (clients, workers, NULL);

    //  我们永远不会到达这里，不过如果因为任何原因到了这里，则执行清理
    zmq_close (clients);
    zmq_close (workers);
    zmq_ctx_destroy (context);
    return 0;
}
```

现在，你应该可以看懂所有的代码了。下面是它的工作原理：

- 服务器启动一组工作线程。每个工作线程创建一个 REP 套接字，然后在此套接字上处理请求。工作线程就像单线程服务器。仅有的区别是传输协议（用 inproc 取代了 tcp）和绑定 - 连接的方向。 67

- 服务器创建一个 ROUTER 套接字去跟客户端交流并将这绑定到它的外部接口（通过 tcp）。
- 服务器创建一个 DEALER 套接字去跟工人交流并将这绑定到它的内部接口（通过 inproc）。
- 服务器启动一个连接两个套接字的代理。代理公平地从所有客户端提取请求并将其分配给那些工人。它还将应答路由回其出发点。

请注意，在大多数编程语言中，创建线程是不可移植的。POSIX 库是 pthreads，但在 Windows 中，你必须使用不同的 API。在我们的例子中，pthread_create() 调用启动一个新的线程来运行我们定义的 worker_routine() 函数。我们将在第 3 章看到如何在一个可移植的 API 中包装这个。

在这里，“工作”只是一秒钟的暂停。不过，我们可以在工人中做任何事情，包括与其他节点交流。图 2-12 显示了从 ØMQ 套接字和节点的角度来看，MT 服务器看起来的样子。请注意，请求 - 应答链是如何成为 REQ-ROUTER-queue-DEALER-REP 的。

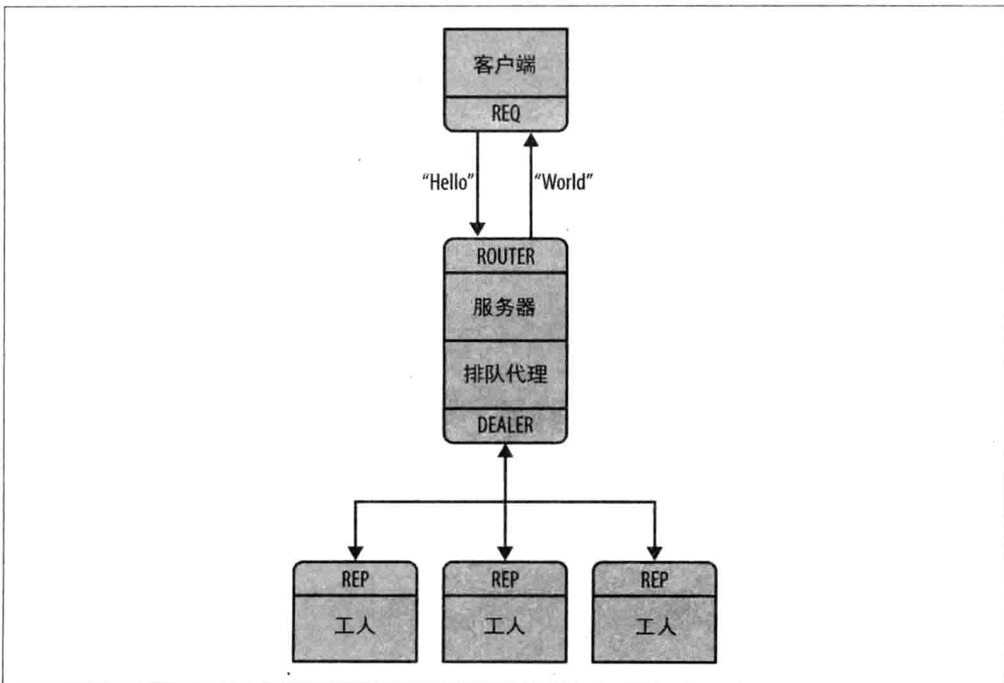


图2-12：多线程服务器

线程间信令（PAIR 套接字） 68

当你开始使用 ØMQ 编写多线程应用程序时，会遇到如何协调线程的问题。虽然你可能会受到诱惑，插入“休眠”语句，或使用诸如信号量或互斥的多线程技术，但你应该使用的唯一机制是 ØMQ 消息。请记住酒鬼和啤酒的故事。

让我们制作三个线程，并使它们准备就绪时彼此发出这样的信号（参见图 2-13）。在示例 2-12 中，我们使用一对套接字在进程内传输。

示例2-12：多线程中继（mtrelay.c）

```
//
//  多线程中继
//
#include "zhelpers.h"
#include <pthread.h>

static void *
step1 (void *context) {
    //  连接到 step2，并告诉它我们已经准备就绪
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step2");
    printf ("Step 1 ready, signaling step 2\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}

static void *
step2 (void *context) {
    //  在启动 step1 前绑定 inproc 套接字
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step2");
    pthread_t thread;
    pthread_create (&thread, NULL, step1, context);

    //  等待信号并传递它
    char *string = s_recv (receiver);
    free (string);
    zmq_close (receiver);

    //  连接到 step3，并告诉它我们已经准备就绪
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step3");
```

```
    printf ("Step 2 ready, signaling step 3\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

69  return NULL;
}

int main (void)
{
    void *context = zmq_ctx_new ();

    //  在启动step2前绑定inproc套接字
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step3");
    pthread_t thread;
    pthread_create (&thread, NULL, step2, context);

    //  等待信号
    char *string = s_recv (receiver);
    free (string);
    zmq_close (receiver);

    printf ("Test successful!\n");
    zmq_ctx_destroy (context);
    return 0;
}
```

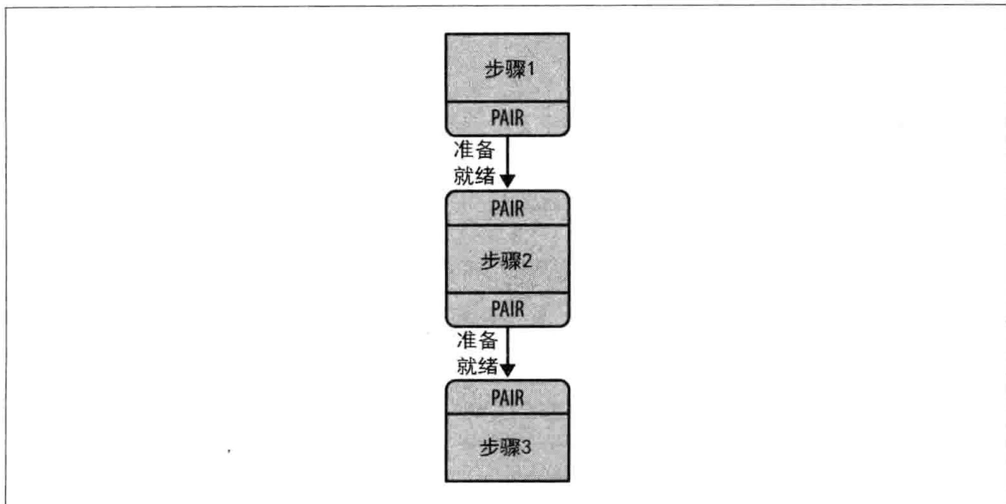


图2-13：接力赛

这是使用 ØMQ 进行多线程编程的一个经典模式：

1. 两个线程通过 inproc 通信，使用的是共享的上下文。
2. 父线程创建一个套接字，将其绑定到一个 inproc 端点，然后启动子线程，将上下文传递给它。
3. 子线程创建第二个套接字，将它连接到该 inproc 端点，然后发信号告诉父线程，它已准备就绪。 70

请注意，使用这种模式的多线程代码是不可扩展到进程的。如果你使用 inproc 和套接字对，你就正在构建一个紧耦合的应用程序，也就是说，其中你的线程在结构上是相互依存的，只有在低延迟真的很重要的时候才这样做。另一种设计模式是一个松耦合的应用程序，其中的线程有自己的上下文并通过 ipc 或 tcp 通信。你可以轻松地将松耦合的线程分解为单独的进程。

这是我们第一次用 PAIR 套接字展示一个例子。为什么要使用 PAIR 呢？其他套接字组合似乎也能够工作，但它们都有副作用，可能会干扰信令：

- 你可以让发送者使用 PUSH 并让接收者使用 PULL。这看起来很简单，并能够正常工作，但请记住，PUSH 将把消息分发到所有存在的接收者。如果你不小心启动了两个接收者（例如，已经有一个正在运行，你又启动了第二个），就会“丢失”一半的信号。PAIR 具有拒绝多个连接的优势，两个连接组成的对是独占的。
- 你可以让发送者使用 DEALER 并让接收者使用 ROUTER。然而，ROUTER 将你的信息包装在一个“封包”中，这意味着你的大小为零的信号变成了一个多部分消息。如果你不关心数据并把任何东西都当作一个有效的信号，并且如果你不会不止一次地从套接字上读入，这并不重要。但是，如果你决定要发送实际数据时，你会突然发现 ROUTER 为你提供了“错误”的消息。DEALER 也分发传出消息，这带来与 PUSH 相同的风险。
- 你可以让发送者使用 PUB，而让接收者使用 SUB。这将完全按照你发送它们的原样正确地传递你的消息，而且 PUB 不像 PUSH 或 DEALER 那样分发消息。但是，你需要用空订阅配置订阅者，这很烦人。更糟的是，PUB-SUB 链接的可靠性是与时间相关的，并且如果在 PUB 套接字发送消息时，SUB 套接字正在连接，信息就有可能会丢失。

这些原因使得 PAIR 成为线程对之间协调的最佳选择。

节点协调

当你想协调节点时，PAIR 套接字就无法正常工作了。这是线程和节点的策略不同的少数

地方之一。原则上，节点可以加入进来或退出去，而线程是静态的。如果远程节点离开后又回来，PAIR 套接字不会自动重新连接。

71 线程和节点之间第二个明显的区别是，你的线程数量通常是固定的，而节点数目是更有可能变化的。让我们以之前的一个场景（天气服务器和客户端）为例，并使用节点协调，以确保订阅者在启动时不会丢失数据。

这个应用程序的工作原理是：

- 发布者事先知道它预计有多少订阅者。这只是一个从某个地方得到的神奇的数字。
- 发布者启动，并等待所有订阅者进行连接。这是节点协调部分。每个订阅者都订阅，然后告诉发布者它已经准备好通过另一个套接字接收数据。
- 当发布者连接了所有订阅者时，它开始发布数据。

在这种情况下，我们将使用一个 REQ-REP 套接字流来同步订阅者和发布者（参见图 2-14）。发布者的代码如示例 2-13 所示。

示例2-13：同步的发布者（syncpub.c）

```
//
//  同步的发布者
//
#include "zhelpers.h"

//  我们等待 10 个订阅者
#define SUBSCRIBERS_EXPECTED  10

int main (void)
{
    void *context = zmq_ctx_new ();

    //  与客户端交流的套接字
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5561");

    //  用于接收信号的套接字
    void *syncservice = zmq_socket (context, ZMQ_REP);
    zmq_bind (syncservice, "tcp://*:5562");

    //  从订阅者获得同步
    printf ("Waiting for subscribers\n");
    int subscribers = 0;
    while (subscribers < SUBSCRIBERS_EXPECTED) {
        //  - 等待同步请求
```

```
        char *string = s_recv (syncservice);
        free (string);
        //  - 发送同步应答
        s_send (syncservice, "");
        subscribers++;
    }
    //  现在广播正好 1M 个更新，后面跟着 END
    printf ("Broadcasting messages\n");
    int update_nbr;
    for (update_nbr = 0; update_nbr < 1000000; update_nbr++)
        s_send (publisher, "Rhubarb");

    s_send (publisher, "END");

    zmq_close (publisher);
    zmq_close (syncservice);
    zmq_ctx_destroy (context);
    return 0;
}
```

72

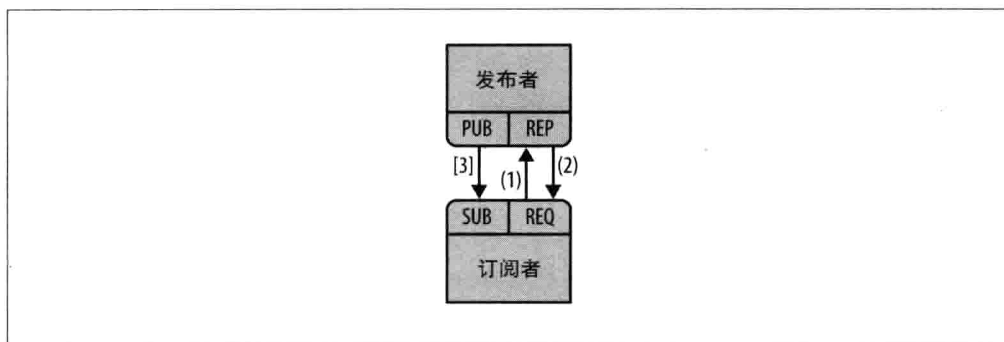


图2-14：发布-订阅同步

订阅者的代码如示例 2-14 所示。

示例2-14：同步的订阅者（syncsub.c）

```
//
//  同步的订阅者
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();
```

```
    //  首先，连接订阅者套接字
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5561");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);

    //  ØMQ 太快了，所以我们要等一段时间 ...
    sleep (1);

    //  其次，与发布者同步
    void *syncclient = zmq_socket (context, ZMQ_REQ);
    zmq_connect (syncclient, "tcp://localhost:5562");
```

73

```
    //  - 发送同步请求
    s_send (syncclient, "");

    //  - 等待同步应答
    char *string = s_recv (syncclient);
    free (string);

    //  第三，获取更新并报告收到的更新数量
    int update_nbr = 0;
    while (1) {
        char *string = s_recv (subscriber);
        if (strcmp (string, "END") == 0) {
            free (string);
            break;
        }
        free (string);
        update_nbr++;
    }
    printf ("Received %d updates\n", update_nbr);

    zmq_close (subscriber);
    zmq_close (syncclient);
    zmq_ctx_destroy (context);
    return 0;
}
```

这个 Bash shell 脚本将启动 10 个订阅者，然后启动发布者：

```
echo "Starting subscribers..."
for ((a=0; a<10; a++)); do
    syncsub &
done
echo "Starting publisher..."
```

```
syncpub
```

这给了我们下面这个令人满意的输出：

```
Starting subscribers...
Starting publisher...
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
Received 1000000 updates
```

我们不能假设 SUB 连接在 REQ/REP 对话完成的时候完成。如果你使用除 `inproc` 外的任何传输工具，就不能保证，出站连接将按照任何顺序完成。所以，这个例子在订阅和 74 发送 REQ/REP 同步之间执行一秒的蛮力休眠。

一个更强大的模式可以是：

- 发布者打开 PUB 套接字并开始发送“Hello”的消息（不是数据）。
- 订阅者连接到 SUB 套接字，当它们收到“Hello”消息时，它们通过 REQ/REP 套接字对告诉发布者。
- 当发布者已经得到所有必要的确认时，它就开始发送实际数据。

零拷贝

ØMQ 的消息 API 可让你直接从应用程序缓冲区发送和接收消息，而不用复制数据。我们把它称为*零拷贝*，并且可以在某些应用程序中用它来提高性能。像所有的优化方法一样，只有当你知道它有帮助，并测量优化前后的效果后才能使用它。零拷贝使你的代码更复杂。

要做到零拷贝，你可以用 `zmq_msg_init_data()` 来创建一个消息，它指向已经用 `malloc()` 在堆上分配的数据块，然后将它传递给 `zmq_msg_send()`。当你创建消息时，你也传递一个函数，当 ØMQ 已经完成传送讯息时，将调用此函数来释放数据块。这是最简单的例子，假设“缓冲区”是在堆上分配的一个 1000 字节的块：

```
void my_free (void *data, void *hint) {
    free (data);
```

```
}
//  发送来自缓冲区的消息，这个缓冲区是我们分配的，而 ØMQ 会帮我们释放它
zmq_msg_t message;
zmq_msg_init_data (&message, buffer, 1000, my_free, NULL);
zmq_msg_send (socket, &message, 0);
```

在接收时没有办法做到零拷贝：ØMQ 传递给你一个缓冲区，你可以一直在其中存储数据，但它不会将数据直接写入应用程序的缓冲区。

在写入方面，ØMQ 的多部分消息能与零拷贝很好地合作。在传统的消息传递中，你就需要调度各个不同的缓冲区，将它们一起放入一个你可以发送的缓冲区。这意味着需要复制数据。使用 ØMQ，你可以将不同来源的多个缓冲区作为单独的消息帧发送，每个字段作为一个用长度分隔的帧发送。对于应用程序，它看起来像一系列的发送和接收调用，但在内部，多个部分是使用单个系统调用写入到网络上并读回的，所以它是非常高效的。

75

发布 - 订阅消息封包

在发布 - 订阅模式中，我们可以将键分割成我们称之为封包（*envelope*）的一个单独的消息帧。如果你想使用发布 - 订阅封包，可以自己建立它们。它是可选的，并且在以前的发布 - 订阅的例子中，我们并没有这么做。在简单的情况下，使用发布 - 订阅封包是多了一点工作，但它更清晰，尤其是对真实案例来说，其中键和数据本质上是不同的东西。

图 2-15 显示的是使用封包的发布 - 订阅消息的样子。

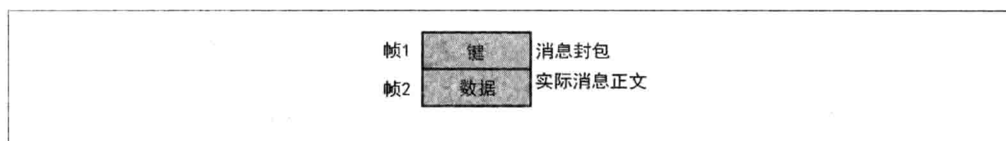


图2-15：带有单独的键的发布-订阅封包

回想一下，订阅执行的是前缀匹配。也就是说，它们会寻找："以 XYZ 开头的所有消息"。最明显的问题是：如何将数据与键分隔开，以免前缀匹配不小心匹配了数据。最好的答案是使用一个封包，因为这个匹配将不会跨越帧边界。

下面是发布 - 订阅封包最简约的代码示例。该发布者（参见示例 2-15）发送两种类型的消息，A 和 B。封包保存了消息类型。

示例2-15：发布-订阅封包发布者（psenvpub.c）

```
//
//  发布 - 订阅封包发布者
//  注意 zhelpers.h 文件也提供了 s_sendmore
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和发布者
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5563");

    while (1) {
        //  编写两个消息，每个消息都带有一个封包和内容
        s_sendmore (publisher, "A");
        s_send (publisher, "We don't want to see this");
        s_sendmore (publisher, "B");
        s_send (publisher, "We would like to see this");
        sleep (1);
    }
    //  我们永远不会到达这里，但如果因为任何原因到达了，就执行清理工作
    zmq_close (publisher);
    zmq_ctx_destroy (context);
    return 0;
}
```

76

示例 2-16 所示的订阅者只希望收到 B 类型的消息。

示例2-16：发布-订阅封包订阅者（psenvsub.c）

```
//
//  发布 - 订阅封包订阅者
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和发布者
    void *context = zmq_ctx_new ();
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5563");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "B", 1);

    while (1) {
```

```
        //  读取封包的地址
        char *address = s_recv (subscriber);
        //  读取消息内容
        char *contents = s_recv (subscriber);
        printf ("[%s] %s\n", address, contents);
        free (address);
        free (contents);
    }
    //  我们永远不会到达这里，但如果因为任何原因到达了，就执行清理工作
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

当你运行这两个程序时，订阅者应该显示如下信息：

```
[B] We would like to see this
[B] We would like to see this
[B] We would like to see this
[B] We would like to see this
...
```

这个例子说明，订阅过滤器拒绝或接受整个多部分消息（键加数据）。你永远都不会得到一个多部分消息的一部分。

如果你订阅了多个发布者，而你想知道它们的地址，以便可以通过另一个套接字发送数据给它们（这是一个典型的用例），那么可以创建一个由三部分组成的消息，如图 2-16 所示。

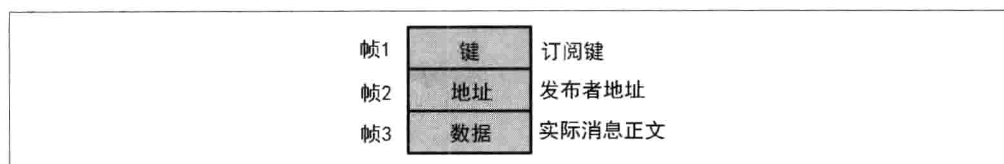


图2-16：带有发送者地址的发布-订阅封包

77

高水位标记

当你能够迅速从进程向进程发送消息时，你很快就会发现，内存是宝贵的资源，并且很容易填满。除非你理解这个问题，并采取预防措施，否则在进程中某处几秒钟的延迟，都可能变成撑爆一个服务器的积压。

问题是这样的：如果有一个进程 A 正发送消息给进程 B，而进程 B 突然变得非常繁忙（执行垃圾收集，CPU 过载，等等），那么进程 A 要发送的消息会发生什么情况呢？有些消息会位于 B 的网络缓冲区，有些消息会位于以太网线路本身，有些消息会位于 A 的网络缓冲区，其余的会积聚在 A 的内存中。如果不采取一些预防措施，A 很容易就会耗尽内存并且崩溃。这是消息代理的一个经典问题。

解决办法是什么？一个办法是将问题交给上游。A 是从别的地方得到的消息，所以可以告诉该进程停下来，排队，这就是所谓的*流量控制*。这听起来不错，但如果你发送的是 Twitter 的微博呢？你能告诉整个世界，在 B 采取行动时停止发微博吗？

流量控制在某些情况下能够工作，而在有些情况下不能工作。传输层也不能叫应用层"停止"，好比地铁系统不可以告诉一个大型企业，"请让你的员工继续工作半小时。我太忙了。"

对于消息传递，解决办法是设置缓冲区的大小限制，然后，当我们到达这些限制时，采取一些明智的行动。在某些情况下（不过不是一个地铁系统），答案是丢弃消息。在其他情况下，最好的策略是等待。

ØMQ 使用高水位标记（*high-water mark*，HWM）的概念来定义其内部管道的容量。每个从套接字出来或连入套接字的连接都有它自己用于发送和 / 或接收的管道和 HWM，这取决于套接字类型。有些套接字（PUB、PUSH）只有发送缓冲区，有些套接字（SUB、PULL、REQ、REP）只有接收缓冲区，有些套接字（DEALER、ROUTER、PAIR）同时具有发送缓冲区和接收缓冲区。

在 ØMQ v2.x 版本中，HWM 默认是无限的。在 ØMQ v3.x 版本，它在默认情况下为 1000，这是比较明智的设置。如果你还在使用 ØMQ v2.x 版本，你应该总是在你的套接字上设置 HWM，无论是设置为 1000 以匹配 ØMQ v3.x，还是设置为另一个考虑到你的消息大小的数字。

当你的套接字达到其 HWM 时，根据不同的套接字类型，这将阻塞或删除数据。如果它 78
们达到其 HWM，PUB 和 ROUTER 套接字将丢弃数据，而其他类型的套接字将阻塞。

通过 `inproc` 传输时，发送者和接收者共享同一个缓冲区，所以真正的 HWM 等于由两侧设置的 HWM 的总和。

最后，高水位标记都用消息部分，而不是整个消息来计算。如果要发送两部分消息，默认 HWM 是 500。如果使用 ROUTER 套接字类型（详见下一章讨论），那么每个消息至少是两部分。

消息丢失问题的解决方案

在使用 ØMQ 编写应用程序时，你会不止一次地遇到这个问题：你期望得到的消息丢失了。我们已经用一张图（参见图 2-17）来遍历产生这种情况的最常见的原因。

如果你在失败要付出昂贵代价的上下文中使用 ØMQ，那么要好好规划以避免失败的发生。首先，构建一个让你学习和测试你的设计的不同方面的原型。对它们施加压力，直到它们崩溃，以让你知道你的设计究竟有多强大。第二，投资于测试。这意味着建立测试框架，以确保你能够获得具有足够计算能力的逼真设置，并抽出宝贵的时间，或借助其他力量，来严肃地进行实际测试。理想的情况是，一个团队写代码，另一个团队努力破坏它。最后，务必让你的组织联系 iMatix 来讨论我们如何能够帮助确保事情正常工作，并且如果它们出故障了，能够迅速被修复。

总之，如果你还没有证明某个架构能在现实条件下工作，那么它很可能会在最糟糕的时候崩溃。

79

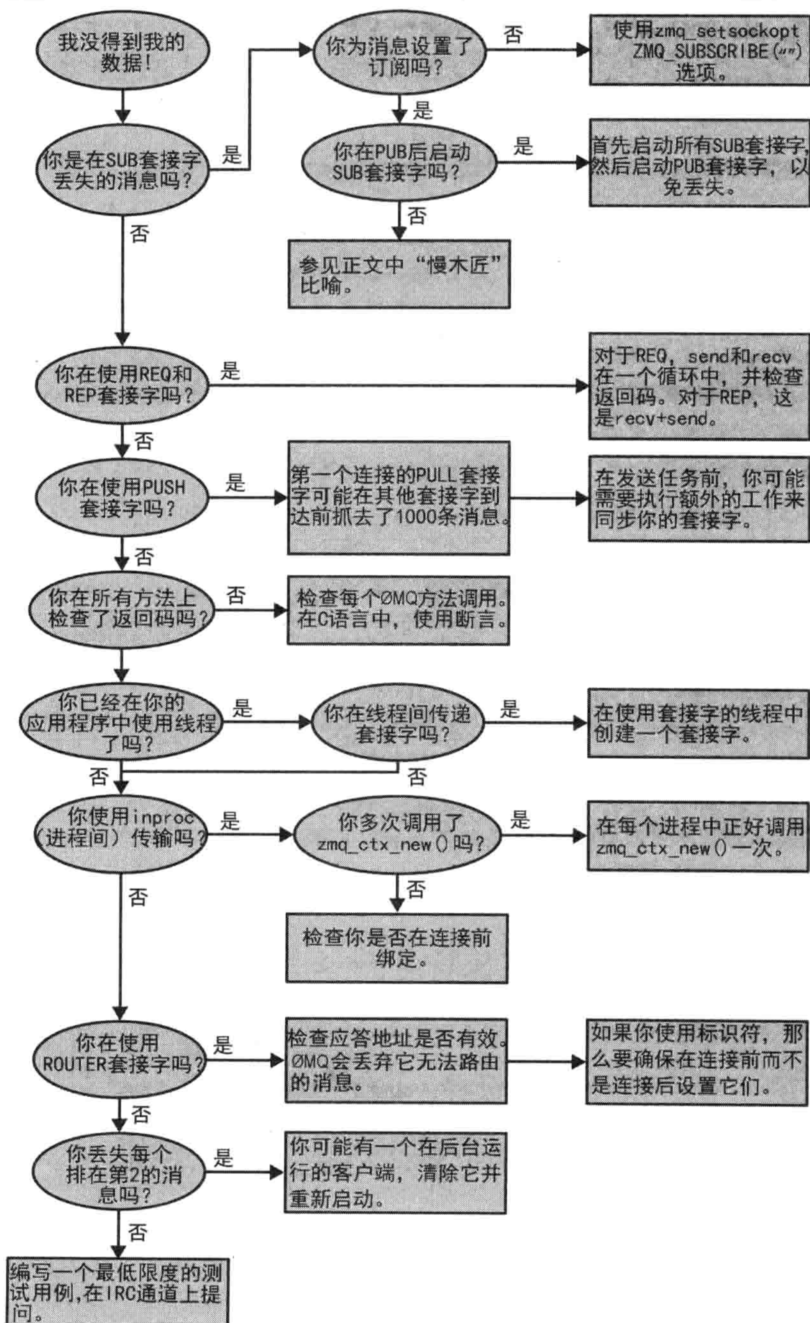


图2-17：消息丢失问题的解决方案

第 3 章 81

高级请求-应答模式

在第 2 章中，我们通过开发一系列的小应用程序，遍历了使用 ØMQ 的基本知识，每次都探索 ØMQ 新的方面。在本章中，我们将继续这种做法，探索建立在 ØMQ 的核心的请求 - 应答模式之上的高级模式。

我们将讨论：

- 请求 - 应答机制的工作原理
- 如何结合 REQ、REP、DEALER 和 ROUTER 套接字
- ROUTER 套接字如何工作的详细原理
- 负载均衡模式
- 构建一个简单的负载均衡消息代理
- 为 ØMQ 设计一个高层次的 API
- 建立一个异步请求 - 应答服务器
- 详细的跨代理路由的例子

请求 - 应答机制

我们已经在多部分消息中简要地介绍过请求 - 应答机制。现在让我们来看一个主要的用例，这是*应答消息封包*（*reply message envelope*）。封包是安全地打包带有地址的数据，而不触及该数据本身的一种方法。通过将应答地址转换成一个封包，我们有可能写出通用的中间层，如创建、读取和删除地址的 API 和代理，无论消息的有效负载或结构是什么。

在请求 - 应答模式中，封包保存应答的返回地址。这是一个没有状态的 ØMQ 网络怎样能创建往返请求 - 应答对话的原因。 82

当你使用 REQ 和 REP 套接字时，你甚至看不到封包，因为这些套接字会自动处理它们。

但对于大多数有趣的请求 - 应答模式，你仍需要了解封包和 ROUTER 套接字。我们将一步一步地继续研究这个。

简单的应答封包

请求 - 应答交换由一个*请求*（*request*）消息和一个最终*应答*（*reply*）消息组成。在简单的请求 - 应答模式中，每个请求都有一个应答。在更高级的模式中，请求和应答可以异步地流动。然而，应答封包总是以同样的方式工作。

该 ØMQ 应答封包正式包含零个或多个应答地址，后跟一个空帧（封包分隔符），其次是消息体（零或多帧）。封包是由在一个链中一起工作的多个套接字创建的。我们接下来将分析它。

我们将以在一个 REQ 套接字中发送“Hello”开始。该 REQ 套接字创建尽可能简单的应答封包，它没有地址，只有一个空的分隔符帧和包含“Hello”字符串的消息帧。这是一个两帧信息（参见图 3-1）。

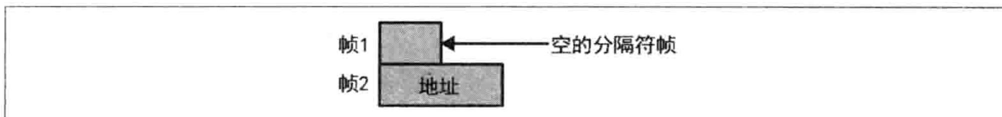


图3-1：带有最简封包的请求

REP 的套接字执行匹配的工作：它剥去封包，直至包括分隔符帧，保存整个封包，并将“Hello”字符串传递给应用程序。因此，我们原来的“Hello World”例子在内部使用了请求 - 应答封包，但应用程序从来没有见过它们。

如果你窥视在 *hwclient* 和 *hwserver* 之间流动的网络数据，那么会看到下面的东西：每个请求和每个应答其实都是两帧，其中一个是空帧，然后是正文。这对于简单的 REQ-REP 对话，似乎没有太大的意义，但是，当我们探讨 ROUTER 和 DEALER 如何处理封包时，你就会看到这么做的原因。

扩展的应答封包

现在让我们通过在中间使用 ROUTER-DEALER 代理来扩展 REQ-REP 对，看看这会如
83 何影响应答封包。这是我们已经在第 2 章看到过的扩展的*请求 - 应答模式*，我们其实可以插入任意多步的代理（参见图 3-2），其中的机制是相同的。

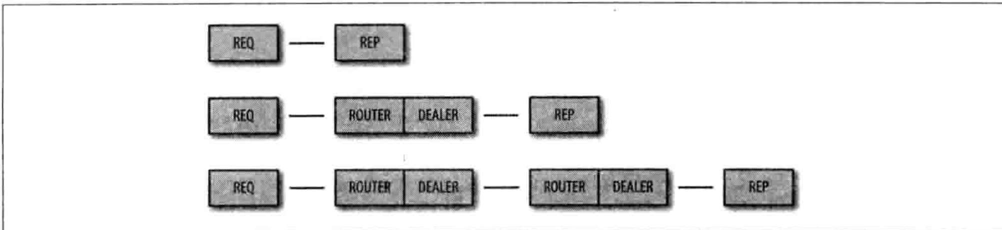


图3-2：扩展的请求-应答模式

代理这样做，用伪代码表示如下：

```
prepare context, frontend and backend sockets
while true:
    poll on both sockets
    if frontend had input:
       read all frames from frontend
       send to backend
    if backend had input:
       read all frames from backend
       send to frontend
```

其含义为：

```
准备上下文、前端和后端套接字
条件为真时：
    轮询两个套接字
    如果前端有输入：
        从前端读取所有的帧
        发送到后端
    如果后端有输入：
        从后端读取所有帧
        发送到前端
```

与其他套接字不同，ROUTER 套接字跟踪它具有的每一个连接，并将这些告诉调用者。它告诉调用者的方法是，在所接收的每个消息的前面黏附连接身份（*identity*）。身份，有时也被称为地址（*address*），仅仅是一个二进制字符串，它除了表示“这是一个针对该连接的独特句柄”，没有任何其他意义。然后，当你通过 ROUTER 接口发送消息时，首先发送一个身份帧。

`zmq_socket()` 手册页是这样描述它的：

> 在接收消息时，ZMQ_ROUTER 套接字在把它传递给应用程序之前，应预置一个包含原始节点消息的身份的消息部分。接收到的消息是在所有连接节点中公平排队的。当发送消息时，ZMQ_ROUTER 套接字应删除消息的第一部分，并用它来确定消息将被路由到的节点的身份。

作为一个历史注记，ØMQ v2.2 及更早版本使用通用唯一标识符（UUID）作为身份，而 ØMQ v3.0 及更高版本使用短整型数，这会对网络性能造成一定影响，但只有当你使用多个代理跃点时才会发生，而这是罕见的。

这个概念理解起来比较困难，但如果你想成为一个 ØMQ 专家，这是必须掌握的。
84 ROUTER 套接字为每个连接发明了一个与之合作的随机的身份。如果有三个 REQ 套接字连接到 ROUTER 套接字，它会发明三个随机的身份，每个 REQ 套接字都有一个身份。

所以，如果我们继续我们能够工作的例子，让我们假设，REQ 套接字具有身份 `02`。在内部，这意味着 ROUTER 套接字维护了一个散列表，可以用它搜索 `02` 并找到该 REQ 套接字的 TCP 连接。

当我们从 ROUTER 套接字收到消息时，会得到三帧（参见图 3-3）。

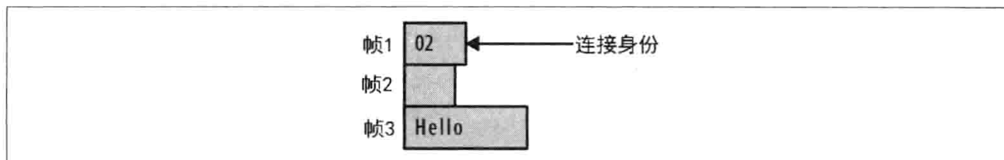


图3-3：带有一个地址的请求

代理循环的核心是“从一个套接字读取，写入另一个套接字”，所以我们最终在 DEALER 套接字上把这些包含三帧的消息发送出去。如果你现在嗅探网络流量，会看到这些三帧消息从 DEALER 套接字飞往 REP 套接字。REP 套接字所做的和以前一样：剥开整个封包，包括新的应答地址，并再次提供“Hello”给调用者。

顺便说一句，REP 套接字同时只能处理一个请求 - 应答交换，这就是为什么如果你尝试读取多个请求或发送多个应答，而没有坚持严格的 recv-send 循环时，它给出一个错误的原因。

你现在应该能够可视化该返回路径。当 `hwserver` 发回“World”时，REP 套接字使用它保存的封包包装它，并跨线路发送一个三帧应答消息给 DEALER 套接字（参见图 3-4）。

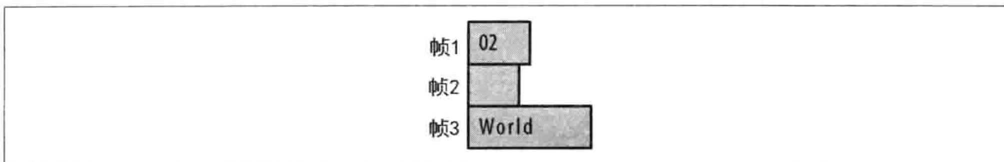


图3-4：带有一个地址的应答

现在 DEALER 读取这三帧，并通过 ROUTER 套接字发送所有三个输出。ROUTER 取得消息的第一帧，这是 02 身份，并查找该连接。如果它找到了对应于该身份的连接，再把接下来的两帧发送到线路中（参见图 3-5）。

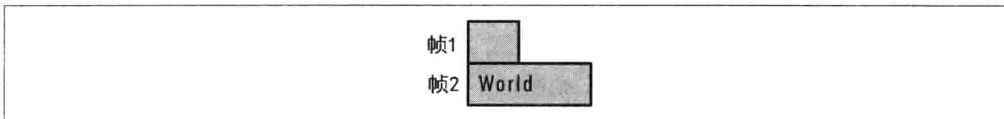


图3-5：带有最简封包的应答

REQ 套接字拾起这个消息，并检查第一帧是否是空分隔符，实际上它是。然后，它会丢 85
弃该帧，并将“World”传递给调用应用程序，应用程序将这打印出来，这使得我们这些第一次看 ØMQ 的人感到惊奇。

这有什么好处呢

说实话，严格的请求 - 应答用例或扩展的请求 - 应答用例都比较有局限性。一方面，没有简单的方法来从由于应用程序代码缺陷导致的服务器崩溃之类的常见故障中恢复（我们将在第 4 章介绍相关内容）。然而，一旦你掌握了这 4 个套接字处理封包的方式，以及它们是如何互相交流的，就可以做出非常有用的东西。我们看到了 ROUTER 如何使用应答封包来决定将应答路由回哪个客户端 REQ 套接字。现在，让我们换一种方式来表达这件事情 ：

- 每次 ROUTER 为你提供一个消息时，它都用一个身份的形式告诉你此消息是从哪个节点来的。
- 当新的节点到达时，你可以使用一个散列表（用身份作为键）来跟踪它们。
- 如果你把身份作为消息的第一帧缀在前面，ROUTER 就会将消息异步地路由到任何连接到它的节点上。

ROUTER 套接字并不关心整个封包，它们对空分隔符一无所知，它们所关心的只是一个身份帧，这可以让它们辨认出将消息发送到哪个连接。

请求 - 应答套接字回顾

让我们来回顾一下：

- REQ 套接字在消息数据的前面向网络发送一个空分隔符帧。REQ 套接字是同步的：它们总是先发送一个请求，然后等待一个应答。REQ 套接字每次与一个节点交流。如果一个 REQ 套接字连接到多个节点，请求会同时分发到每个节点，并反过来预期从每个节点收到一个应答。
- REP 套接字读取和保存所有的身份帧也包括空分隔符，然后将接下来的帧或多帧传
86 递给调用者。REP 套接字是同步的，每次跟一个节点交流。如果一个 REP 套接字连接到多个节点，那么它会用公平的方式读取请求，并且应答总是被发送到发出最后一个请求的同一个节点。
- DEALER 套接字对应答封包一无所知，并像处理任何多部分消息那样处理它。DEALER 套接字是异步的，就像 PUSH 和 PULL 的结合。它们将发送的消息分发给所有连接，并对从所有连接收到的消息公平地进行排队。
- 与 DEALER 一样，ROUTER 套接字对应答封包也一无所知。它为它的连接创建身份并把这些身份作为任何接收到的消息的第一帧传递给调用者。反之，当调用者发送一条消息时，它使用第一个消息帧作为一个身份来查找要发送到的连接。ROUTER 是异步的。

请求 - 应答组合

我们有 4 个请求 - 应答套接字，每一个套接字都有特定的行为。我们已经在简单的和扩展的请求 - 应答模式中看到它们是如何连接的。但这些套接字是可以用来解决很多问题的构件。

下面这些都是合法的组合：

- REQ 到 REP
- DEALER 到 REP
- REQ 到 ROUTER
- DEALER 到 ROUTER
- DEALER 到 DEALER
- ROUTER 到 ROUTER

而这些组合是无效的（我会解释为什么）：

- REQ 到 REQ

- REQ 到 DEALER
- REP 到 REP
- REP 到 ROUTER

下面是记忆这些语义的一些提示。DEALER 就像是一个异步的 REQ 套接字，而 ROUTER 就像是一个异步的 REP 套接字。在可以使用 REQ 套接字的地方，也可以使用 DEALER，只需要自己读取和写入封包。在可以使用 REP 套接字的地方，可以使用 ROUTER，只需要管理自己的身份。

将 REQ 和 DEALER 套接字视为“客户端”,而将 REP 和 ROUTER 套接字视为“服务器”。 87
大多数情况下，你要绑定 REP 和 ROUTER 套接字，并将 REQ 和 DEALER 套接字连接到它们。事情并不总像这个一样简单，但对于起步阶段，这是一个干净的和令人难忘的地方。

REQ 到 REP 组合

我们已经介绍了 REQ 客户端与 REP 服务器交流，但在这里有一个重要的方面需要提到：REQ 的客户端必须初始化消息流。一个 REP 服务器不能跟一个没有先发出请求的 REQ 客户端交流。从技术上讲，它甚至是不可能的，如果你尝试这么做，API 也将返回一个 EFSM 的错误。

DEALER 到 REP 组合

现在，让我们将 REQ 客户端替换为 DEALER。这给了我们一个可以与多个 REP 服务器交流的异步客户端。如果使用 DEALER 改写“Hello World”客户端，我们就能够发送任何数量的“Hello”请求，而无须等待应答。

当使用 DEALER 跟一个 REP 套接字交流时，我们必须准确地模拟 REQ 套接字会发送的封包，否则 REP 套接字将认为该消息无效而丢弃它。因此，要发送消息，需要采取以下步骤：

1. 发送一个设置了 MORE 标志的空消息帧。
2. 发送消息正文。

而当我们收到一条消息时，应该：

1. 接收第一帧，如果它不是空的，则丢弃整个消息。
2. 接收下一帧并将其传递给应用程序。

REQ 到 ROUTER 组合

同样，我们可以用 DEALER 来替换 REQ，也可以用 ROUTER 来取代 REP。这给了我们一个可以同时与多个 REQ 客户端交流的异步服务器。如果使用 ROUTER 改写“Hello World”服务器，我们就能够并行处理任意数量的“Hello”请求。如我们在第 2 章的 *mtserver* 例子中所看到的。

我们可以用两种不同的方式使用 ROUTER：

- 作为一个在前端和后端套接字之间切换消息的代理。
- 作为一个读取消息并作用于它的应用程序。

88 在第一种情况下，ROUTER 只是简单地读取所有的帧，包括人造的身份帧，并盲目地将它们传递。在第二种情况下，ROUTER 必须知道它正在发送的应答封包的格式。如果另一个节点是一个 REQ 套接字，那么 ROUTER 获取身份帧、一个空帧，然后是数据帧。

DEALER 到 ROUTER 组合

现在，我们可以用 DEALER 和 ROUTER 同时替换 REQ 和 REP 两者，以获得最强大的套接字组合，这是 DEALER 与 ROUTER 交流的组合。它为我们提供了与异步服务器交流的异步客户端，双方都对消息格式拥有完全控制权。

因为 DEALER 和 ROUTER 都可以处理任意的消息格式，所以如果你希望安全地使用这些套接字，必须在某种程度上成为一个协议的设计师。最起码，你必须决定是否要模拟 REQ/REP 应答封包（这取决于你是否真的需要发送应答。）

DEALER 到 DEALER 组合

你可以用 ROUTER 替换 REP，但如果 DEALER 只与一个节点交流，你也可以用 DEALER 替换 REP。

当将一个 REP 替换为 DEALER 时，你的工人会突然成为完全异步的，它发回任意数量的应答。这么做的代价是，你必须自己管理应答封包，并正确地得到它们，否则什么都不会工作。后面我们将看到一个工作示例。远的不说，现在只需要知道 DEALER 到 DEALER 是很难正确使用的模式之一，并庆幸我们很少会需要它就可以了。

ROUTER 到 ROUTER 组合

对于 N 对 N 的连接，这听起来很完美，但它是最难使用的组合。你应该避免使用它，除非你成为 ØMQ 高手。我们将在第 4 章的自由模式中看到它的一个例子，并在第 8 章看

到一个设计用于对等网络工作的替代 DEALER 到 ROUTER 的例子。

无效组合

大多数情况下，尝试将客户端连接到客户端，或将服务器连接到服务器，都是一个坏主意，并将无法正常工作。我会详细解释下面的每个组合都有什么问题，而不是给出含糊的一般性警告。

REQ 到 REQ

双方都希望通过发送消息给对方来启动，而只有在定时所有东西的情况下，使得两个节点在完全相同的时间交换消息，这种组合才能正常工作。甚至考虑这件事都会伤害我的大脑。

REQ 到 DEALER 89

理论上你可以做到这一点，但是如果你添加了第二个 REQ，因为 DEALER 没有办法将应答发送到原来的节点，所以它将被破坏。因此，REQ 套接字会被弄糊涂，并/或返回目的地的是另一个客户端的消息。

REP 到 REP

双方都将等待对方发送第一条消息。

REP 到 ROUTER

在理论上，ROUTER 套接字可以启动对话并发送一个适当格式化的请求，如果它知道 REP 套接字已连接，并且它知道该连接的身份。尽管如此，但这是混乱的，并没有在 DEALER 到 ROUTER 上面增加什么东西。

与无效组合相比，这些有效组合的共同点是，ØMQ 套接字连接总是偏向绑定到一个端点的节点，而另一个连接到该节点。此外，哪一侧绑定和哪一侧连接并不是任意的，而是要遵循自然模式。在我们希望“在那里”的一侧绑定：它会是一台服务器、一个代理、发布者、收集器。在“进来和出去”的一侧连接：它会是客户端或工人。记住这一点将有助于你设计出更好的 ØMQ 架构。

探索 ROUTER 套接字

让我们来更进一步地研究 ROUTER 套接字。我们已经看到，它们通过把不同的信息路由到特定的连接来工作。下面更详细地解释如何识别这些连接，以及当 ROUTER 套接字不能发送消息时，它会做什么。

身份和地址

ØMQ 中的*身份*（*identity*）概念专指 ROUTER 套接字，以及它们如何识别它们到其他套接字的连接。更广泛地说，在应答封包中，身份被用作地址。在大多数情况下，身份是任意的，并且对 ROUTER 套接字是本地的：它是一个散列表中的查找键。独立的，一个节点可以有一个物理地址（如“tcp://192.168.55.117:5670”这样一个网络端点）或逻辑地址（一个 UUID 或电子邮件地址或其他唯一键）。

使用 ROUTER 套接字与特定节点交流的应用程序可以将一个逻辑地址转换为身份，如果它已经建立了必要的散列表。因为当该节点发送消息时，ROUTER 套接字只公布一个连接的身份（到一个特定的节点），你真的只能应答该消息，而不能自发地跟一个节点交流。

即使你翻转这个规则，让 ROUTER 连接到节点，而不是等待节点连接到 ROUTER，这也是成立的。

但是，你可以强制 ROUTER 套接字使用逻辑地址代替其身份。该 zmq_setsockopt() 参考页面把这称作*设置套接字身份*（*setting the socket identity*）。它的工作原理如下：

90

- 节点应用程序在绑定或连接前，设置其节点套接字（DEALER 或 REQ）的 ZMQ_IDENTITY 选项。
- 通常情况下，节点随后连接到已经绑定的 ROUTER 套接字，但 ROUTER 也可以连接到节点。
- 在连接时，节点套接字告诉 ROUTER 套接字，“对于这个连接，请用这个身份。”
- 如果节点套接字不指出这个，那么 ROUTER 会为连接生成其一贯的任意的随机身份。
- ROUTER 套接字现在把这个逻辑地址作为来自那个节点的任何消息的一个前缀身份帧提供给应用程序。
- ROUTER 还期望这个逻辑地址作为发往那个节点的任何传出消息的前缀标识帧。

示例 3-1 是一个简单的连接到 ROUTER 套接字的两节点的例子，其中一个节点强加了逻辑地址“PEER2”。

示例3-1：*身份检查*（identity.c）

```
//
//  演示请求 - 应答模式所使用的身份。运行此程序本身。
//  请注意，实用函数 s_ 由 zhelpers.h 提供。
//  让大家不断重复这个代码很无聊。
//
#include "zhelpers.h"
```

```
int main (void)
{
    void *context = zmq_ctx_new ();

    void *sink = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (sink, "inproc://example");

    //  首先允许 ØMQ 设置身份
    void *anonymous = zmq_socket (context, ZMQ_REQ);
    zmq_connect (anonymous, "inproc://example");
    s_send (anonymous,"ROUTER uses a generated UUID");
    s_dump (sink);

    //  然后设置我们自己的身份
    void *identified = zmq_socket (context, ZMQ_REQ);
    zmq_setsockopt (identified, ZMQ_IDENTITY, "PEER2", 5);
    zmq_connect (identified, "inproc://example");
    s_send (identified, "ROUTER socket uses REQ's socket identity");
    s_dump (sink);

    zmq_close (sink);
    zmq_close (anonymous);
    zmq_close (identified);
    zmq_ctx_destroy (context);
    return 0;
}
```

91

下面是程序打印的输出：

```
----------------------------------------
[005] 006B8B4567
[000]
[026] ROUTER uses a generated UUID
----------------------------------------
[005] PEER2
[000]
[038] ROUTER uses REQ's socket identity
```

ROUTER 错误处理

ROUTER 套接字对它们不能向任何地方发送的消息的处理方式确实有点粗暴：它们默默地丢弃这些消息。在工作代码中，这种态度是有道理的，但它使调试变得困难。“发送身份作为第一帧”的做法太刁钻，当我们正在学习时，会经常遇到这个错误，而当我们陷入困境时，ROUTER 的石头般的沉默不是非常有建设性的。

从 ØMQ v3.2 开始，可以设置一个套接字选项来捕获这个错误：ZMQ_ROUTER_MANDATORY。在 ROUTER 套接字上设置它，并且当你在一个发送调用上提供了一个不可路由的身份时，该套接字将发出信号表明发生 EHOSTUNREACH 错误。

负载均衡模式

现在让我们来查看一些代码。我们将看到如何将 ROUTER 套接字连接到 REQ 套接字，然后再连接到 DEALER 套接字。这两个例子遵循相同的逻辑，这个逻辑是一个*负载均衡*模式。当特意使用 ROUTER 套接字来路由，而不是将其仅仅作为一个应答通道时，这种模式是我们第一次接触到的。

负载均衡模式是很常见的，在本书中，我们会多次看到它。它解决了简单的循环路由（如 PUSH 和 DEALER 提供的）的主要问题，这个问题是，如果任务没有全部都用大致相同的时间来执行，循环就会变得效率低下。

用邮局的例子来做比喻：如果每个柜台都有一个队列，有一些人买邮票（一个快速、简单的交易），而有些人开设新的账户（一个非常缓慢的交易），你会发现，邮票的买家越来越不公平地被卡在队列中。而且，正如在一个邮局中，如果你的消息传递机制是不公平的，人们会生气。

邮局的解决方案是创建单个的队列，这样，即使一个或两个柜台被卡在缓慢的工作上，其他的柜台将继续按照先来先服务的原则为客户提供服务。

92 PUSH 和 DEALER 使用这种简单方法纯粹是因为性能。如果你到达任何美国主要机场，会发现人们在等待移民入境处排长队。边境巡逻官员将把排在队伍前面的人分配给每个柜台，而不是使用一个单一的队列。让人们事先走 50 码为每个乘客节约一两分钟。而且，由于每个护照检查过程大致花费相同的时间，它或多或少是公平的。这是用于 PUSH 和 DEALER 的战略：将发送工作负载的时间提前，以便有较少的旅行距离。

在使用 ØMQ 时，这是一个反复出现的主题：世界上的问题是多种多样的，你能真正地从以正确的方式解决每一个不同的问题中受益。机场不是邮局，并且“一种尺寸”不会真的很好地适合任何人。

让我们返回把代理（ROUTER）连接到工人（DEALER 或 REQ）的情景。该代理必须知道什么时候工人已经准备就绪，并保持一份工人的名单，以便它可以每次取出最近最少使用（*least recently used*）的工人来执行工作。

实际上，解决方案是非常简单的：当工人启动时，以及它们完成每一项任务之后，它们都会发出“准备就绪”的消息。代理一个接一个地读取这些消息。每一次读取一个消息，

这个消息来自最后使用的工人。而且，由于使用的是 ROUTER 套接字，我们可以得到一个身份，就可以用它来将任务发回给该工人。

它是对请求 - 应答的一种扭曲，因为任务是和应答一起被发送的，并且针对任务的任何响应都作为一个新的请求发送。下面的代码示例应该清楚地说明了这件事。

ROUTER 代理和 REQ 工人

示例 3-2 是使用 ROUTER 代理与一系列 REQ 工人交流的负载均衡模式的一个例子。

示例3-2：ROUTER到REQ（rtreq.c）

```
//
//  ROUTER 到 REQ 示例
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_WORKERS 10

static void *
worker_task (void *args)
{
    void *context = zmq_ctx_new ();
    void *worker = zmq_socket (context, ZMQ_REQ);
    s_set_id (worker);          //  设置一个可打印的身份
    zmq_connect (worker, "tcp://localhost:5671");

    int total = 0;
    while (1) {
        //  告诉代理我们已准备好执行工作
        s_send (worker, "Hi Boss");

        //  从代理获取工作负载，直到完成
        char *workload = s_recv (worker);
        int finished = (strcmp (workload, "Fired!") == 0);
        free (workload);
        if (finished) {
            printf ("Completed: %d tasks\n", total);
            break;
        }
        total++;

        //  执行某些随机的工作
        s_sleep (randof (500) + 1);
```

93

```
    }
    zmq_close (worker);
    zmq_ctx_destroy (context);
    return NULL;
}
```

虽然此示例在单个进程中运行，但这只是为了使其更易于启动和停止。每个线程都有自己的上下文并且从概念上可作为一个单独的进程。示例 3-3 显示了主任务。

示例3-3：ROUTER到REQ（rtreq.c）：主任务

```
int main (void)
{
    void *context = zmq_ctx_new ();
    void *broker = zmq_socket (context, ZMQ_ROUTER);

    zmq_bind (broker, "tcp://*:5671");
    srandom ((unsigned) time (NULL));

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_task, NULL);
    }
    //  运行 5 秒钟，然后告诉工人结束
    int64_t end_time = s_clock () + 5000;
    int workers_fired = 0;
    while (1) {
        //  下一个消息给我们最近最少使用的工人
        char *identity = s_recv (broker);
        s_sendmore (broker, identity);
        free (identity);
        free (s_recv (broker));     //  封包分隔符
        free (s_recv (broker));     //  来自工人的响应
        s_sendmore (broker, "");

94      //  鼓励工人，除非到了解雇它们的时候
        if (s_clock () < end_time)
            s_send (broker, "Work harder");
        else {
            s_send (broker, "Fired!");
            if (++workers_fired == NBR_WORKERS)
                break;
        }
    }
    zmq_close (broker);
```

```
    zmq_ctx_destroy (context);
    return 0;
}
```

该示例运行 5 秒钟，然后每个工人都输出它处理了多少任务。如果路由正常工作，我们会期望工作被公平地分配：

```
Completed: 20 tasks
Completed: 18 tasks
Completed: 21 tasks
Completed: 23 tasks
Completed: 19 tasks
Completed: 21 tasks
Completed: 17 tasks
Completed: 17 tasks
Completed: 25 tasks
    Completed: 19 tasks
```

要与这个例子中的工人交流，我们要创建一个身份加一个空封包分隔符帧组成的 REQ 友好型封包（参见图 3-6）。

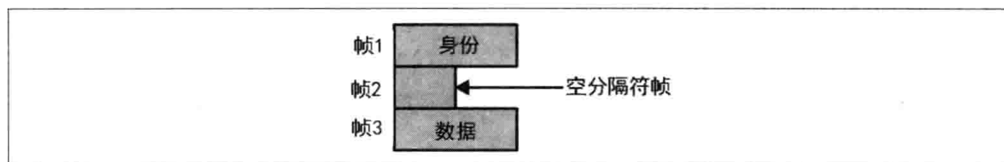


图3-6：用于REQ的路由封包

ROUTER 代理及 DEALER 工人

在任何可以使用 REQ 的地方，都可以使用 DEALER。它们之间有两个特定的差异：

- REQ 套接字总是在发送任何数据帧之前发送一个空分隔符帧，而 DEALER 不这么做。
- REQ 套接字在收到应答之前，将只发送一个消息，而 DEALER 是完全异步的。

对于我们的例子，因为我们正在执行严格的请求 - 应答，所以同步与异步行为无影响。 95
当我们要从失败中恢复时，这是更相关的，将在第 4 章讨论这个内容。

现在让我们来看完全相同的例子，但将 REQ 套接字替换为 DEALER 套接字（参见示例 3-4）。

示例3-4：ROUTER到DEALER（rtdealer.c）

```
//
//  ROUTER 到 DEALER 示例
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_WORKERS 10

static void *
worker_task (void *args)
{
    void *context = zmq_ctx_new ();
    void *worker = zmq_socket (context, ZMQ_DEALER);
    s_set_id (worker);          //  设置一个可打印的身份
    zmq_connect (worker, "tcp://localhost:5671");

    int total = 0;
    while (1) {
        //  告诉代理我们已准备好执行工作
        s_sendmore (worker, "");
        s_send (worker, "Hi Boss");

        //  从代理获取工作负载，直到完成
        free (s_recv (worker));     //  封包分隔符
        char *workload = s_recv (worker);
...
```

该代码与修改前的代码几乎是相同的，不同之处在于，工人现在使用一个 DEALER 套接字，并且读取和写入数据帧之前的空帧。在我们想保持与 REQ 工人的兼容性的时候，应该使用这个方法。

但是，请记住使用该空分隔符帧的原因：它允许在一个 REP 套接字中终止多个跃点扩展的请求，这使用分隔符来分隔应答封包，以便它可以将数据帧传递到它的应用程序中。

如果永远不需要单独传递消息给一个 REP 套接字，我们可以简单地在两侧删除空分隔符帧，这让事情变得更简单。这种设计通常是用于纯 DEALER 到 ROUTER 的协议的。

96 负载均衡的消息代理

前面的例子只完成了一半任务。它可以用假请求和应答管理一组工人，但没有办法去跟客户端交流。

如果再加上第二个接受客户端请求的前端 ROUTER 套接字，并将我们的例子转换成一

个可以将消息从前端切换至后端的代理，我们就得到了一个有用的和可重用的小型负载均衡消息代理（参见图 3-7）。

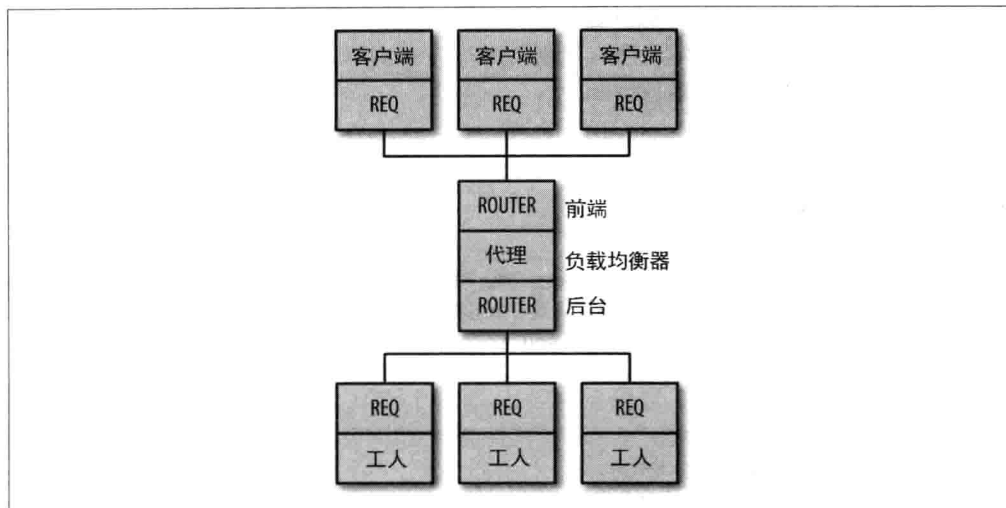


图3-7：负载均衡代理

该代理执行以下操作：

- 接受来自一组客户端的连接
- 接受来自一组工人的连接
- 接受来自客户端的请求，并在一个队列中保存这些请求
- 使用负载均衡模式将这些请求发送给工人
- 接收工人发回来的应答
- 将这些应答发送回原来请求的客户端

该代理的源代码（在示例 3-5 中列出）相当长，但值得去好好理解。

示例3-5：负载均衡代理（lbbroker.c）

```
//
//  负载均衡代理
//  这里显示在进程中的客户端和工人
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
```

97

```
//  对实现为任何东西的数组的队列执行出队操作
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) - sizeof (q [0]))

//  使用 REQ 套接字的基本请求 - 应答客户端。
//  由于 s_send 和 s_recv 不能处理 ØMQ 二进制身份，
//  我们设置一个可打印的文本身份以供路由。
//
static void *
client_task (void *args)
{
    void *context = zmq_ctx_new ();
    void *client = zmq_socket (context, ZMQ_REQ);
    s_set_id (client);          //  设置一个可打印的身份
    zmq_connect (client, "ipc://frontend.ipc");

    //  发送请求，获取应答
    s_send (client, "HELLO");
    char *reply = s_recv (client);
    printf ("Client: %s\n", reply);
    free (reply);
    zmq_close (client);
    zmq_ctx_destroy (context);
    return NULL;
}
```

虽然这个例子在单个进程中运行，但这只是为了使它能更方便地启动和停止。每个线程都有自己的上下文并且在概念上作为一个单独的进程。示例 3-6 显示了工人的任务，使用 REQ 套接字做负载均衡。由于 s_send() 和 s_recv() 不能处理 ØMQ 二进制身份，因此我们设置了可打印文本身份以允许路由。

示例3-6：负载均衡代理（lbbroker.c）：工人的任务

```
static void *
worker_task (void *args)
{
    void *context = zmq_ctx_new ();
    void *worker = zmq_socket (context, ZMQ_REQ);
    s_set_id (worker);          //  设置一个可打印的身份
    zmq_connect (worker, "ipc://backend.ipc");

    //  告诉代理我们已准备好执行工作
    s_send (worker, "READY");

    while (1) {
```

```
        // 读取并保存所有帧，直到我们得到一个空帧
        // 虽然在本例中只有一帧，但可以有多帧
        char *identity = s_recv (worker);
        char *empty = s_recv (worker);
        assert (*empty == 0);
        free (empty);

        // 获取请求，发送应答
        char *request = s_recv (worker);
        printf ("Worker: %s\n", request);
        free (request);

        s_sendmore (worker, identity);
        s_sendmore (worker, "");
        s_send     (worker, "OK");
        free (identity);
    }
    zmq_close (worker);
    zmq_ctx_destroy (context);
    return NULL;
}
```

98

主任务启动客户端和工人，然后路由两层之间的请求（参见示例3-7）。当它们启动时，工人发出信号“准备就绪”，在那之后，当它们使用返回到客户端的一个响应来应答时，我们把它们当作已准备就绪。负载均衡的数据结构只是下一个可用的工人的一个队列。

示例3-7：负载均衡代理（lbbroker.c）：主任务

```
int main (void)
{
    // 准备上下文和套接字
    void *context = zmq_ctx_new ();
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (frontend, "ipc://frontend.ipc");
    zmq_bind (backend,  "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++) {
        pthread_t client;
        pthread_create (&client, NULL, client_task, NULL);
    }
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_task, NULL);
    }
```

示例 3-8 显示了最近最少使用的队列中的主循环。它有两个套接字：一个前端的客户端和一个后端的工人。它在所有情况下轮询后端，并只在有一个或多个工人准备就绪时才
99 轮询前端。这是使用 ØMQ 自己的队列来保存我们还没有准备好处理的消息的一个整洁的方式。当得到一个客户端的应答时，我们弹出下一个可用的工人，将请求发送给它，包括原始客户端的身份。当一名工人应答时，我们将该工人重新排队，并使用应答封包将此应答转发给原来的客户端。

示例3-8：负载均衡代理（lbbroker.c）：主任务程序体

```
// 可用的工人的队列
int available_workers = 0;
char *worker_queue [10];

while (1) {
    zmq_pollitem_t items [] = {
        { backend,  0, ZMQ_POLLIN, 0 },
        { frontend, 0, ZMQ_POLLIN, 0 }
    };
    // 只有在我们有可用的工人时才轮询前端
    int rc = zmq_poll (items, available_workers ? 2 : 1, -1);
    if (rc == -1)
        break;              // 中断

    // 在后台处理工人活动
    if (items [0].revents & ZMQ_POLLIN) {
        // 对工人身份排队以用于负载均衡
        char *worker_id = s_recv (backend);
        assert (available_workers < NBR_WORKERS);
        worker_queue [available_workers++] = worker_id;

        // 第 2 帧是空的
        char *empty = s_recv (backend);
        assert (empty [0] == 0);
        free (empty);

        // 第 3 帧是 READY，或者是一个客户端应答身份
        char *client_id = s_recv (backend);

        // 如果客户端应答了，则把剩余的发回给前端
        if (strcmp (client_id, "READY") != 0) {
            empty = s_recv (backend);
            assert (empty [0] == 0);
            free (empty);
            char *reply = s_recv (backend);
```

```
                s_sendmore (frontend, client_id);
                s_sendmore (frontend, "");
                s_send     (frontend, reply);
                free (reply);
                if (--client_nbr == 0)
                    break;      //  在N个消息后退出
            }
            free (client_id);
        }
```

示例 3-9 显示了我们如何处理客户端请求。 100

示例3-9：负载均衡代理（lbbroker.c）：处理客户端请求

```
        if (items [1].revents & ZMQ_POLLIN) {
            //  现在获取下一个客户端请求，路由到最后使用的工人
            //  客户端请求是 [ 身份 ][ 空 ][ 请求 ]
            char *client_id = s_recv (frontend);
            char *empty = s_recv (frontend);
            assert (empty [0] == 0);
            free (empty);
            char *request = s_recv (frontend);

            s_sendmore (backend, worker_queue [0]);
            s_sendmore (backend, "");
            s_sendmore (backend, client_id);
            s_sendmore (backend, "");
            s_send     (backend, request);

            free (client_id);
            free (request);

            //  将下一个工人身份出队并删除
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            available_workers--;
        }
    }
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

这一方案的难点在于每个套接字读取和写入的封包和负载均衡算法。我们将依次介绍这些，先从消息封包的格式开始。

让我们遍历一个完整的从客户端到工人并返回的请求 - 应答链。在这段代码中，我们设置客户端和工人套接字的身份，以使其更容易追踪消息帧。在实际工作中，我们会允许 ROUTER 套接字发明身份来进行连接。假设客户端的身份是“CLIENT”，工人的身份是“WORKER”。客户端应用程序发送一个包含“HELLO”的单个的帧（参见图 3-8）。



图3-8：客户端发送的消息

101 由于 REQ 套接字增加了其空分隔符帧而 ROUTER 套接字增加了其连接身份，代理从前端 ROUTER 套接字读出三帧：客户端地址、空分隔符帧，以及数据部分（参见图 3-9）。

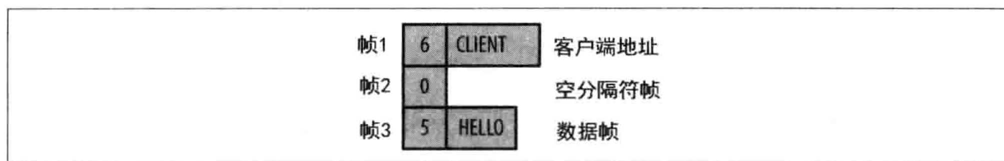


图3-9：前端传入的消息

代理将其发送给工人，前面缀上选定的工人地址，再加上一个额外的空白部分，以使在另一端的 REQ 保持满意（参见图 3-10）。

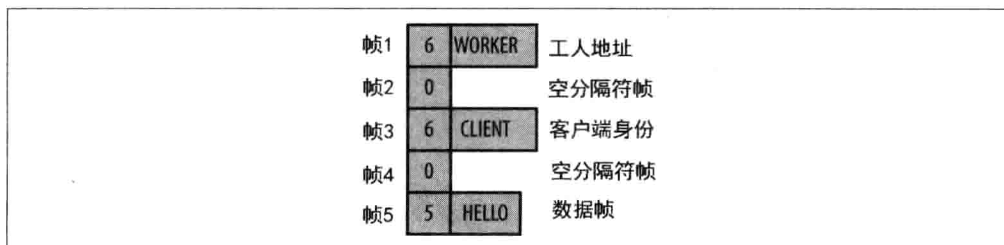


图3-10：发送到后端的消息

这种复杂的封包，首先由后端 ROUTER 套接字消化，从而消除了第一个帧。然后在工人的 REQ 套接字中移除空白部分，并将其余的内容提供给工人应用程序（参见图 3-11）。

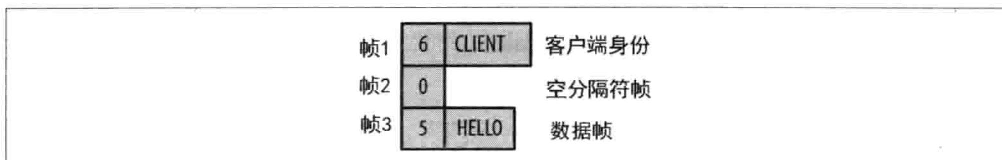


图3-11：传递给工人的消息

工人必须保存封包（也就是包括空消息帧的所有部分），然后它可以对数据部分做必要的处理。请注意，REP 套接字会自动做到这一点，但我们使用的是 REQ-ROUTER 模式，可以得到适当的负载均衡。

在返回的路径上，消息都与它们进来时是一样的，也就是说，后端套接字传给代理的一 102
个消息分为 5 个部分，代理发送给前端套接字的消息分为 3 个部分，而客户端在一个部分内得到 1 个消息。

现在让我们看看负载均衡算法。它要求客户端和工人两者都使用 REQ 套接字，并且要求工人正确地存储和回放它们得到的消息的封包。该算法是：

- 创建一个轮询集，它始终轮询后端，并仅当有一个或多个工人可用时，才轮询前端。
- 使用无限超时来轮询活动。
- 如果在后端有活动，我们要么有一个“准备就绪”的消息，要么有来自客户端的应答。在任一种情况下，我们在工作队列中存储工人地址（第一部分），并且如果其余的内容是客户端应答，我们就通过前端将它发送回客户端。
- 如果在前端有活动，我们取得客户端的请求，弹出下一个工人（这是最后一次使用的），并将请求发送到后端。这意味着发送工人地址、空的部分，然后是客户端请求的 3 个部分。

现在应该看到，你可以基于工人在其最初的“准备就绪”消息中提供的信息来重用和扩展负载均衡算法的变体。例如，工人可能会启动起来，执行一个性能自我测试，然后告诉代理，它们的速度有多快。那么代理就可以选择可用的最快的工人，而不是最早启动的工人。

用于 ØMQ 的一个高级别的 API

现在，我们要将请求 - 应答推入堆栈并打开一个不同的区域，这就是 ØMQ API 本身。走这个弯路还有一个原因：当我们编写更复杂的例子时，低级别的 ØMQ API 开始显得越来越笨拙。下面来看负载均衡代理的工人线程的核心：

```
while (true) {
```

```
        // 读取并保存所有帧，直到我们得到一个空帧
        // 虽然在本例中只有一帧，但可以有多帧
        char *address = s_recv (worker);
        char *empty = s_recv (worker);
        assert (*empty == 0);
        free (empty);

        // 获取请求，发送应答
        char *request = s_recv (worker);
        printf ("Worker: %s\n", request);
        free (request);

        s_sendmore (worker, address);
        s_sendmore (worker, "");
103     s_send     (worker, "OK");
        free (address);
    }
```

该代码甚至是无法重复使用的，因为它只能在封包中处理一个应答地址，并且它已经在ØMQ API的外边做了一些包装。如果直接使用 `libzmq` API，我们不得不编写如下代码：

```
    while (true) {
        // 读取并保存所有帧，直到我们得到一个空帧
        // 虽然在本例中只有一帧，但可以有多帧
        zmq_msg_t address;
        zmq_msg_init (&address);
        zmq_msg_recv (worker, &address, 0);

        zmq_msg_t empty;
        zmq_msg_init (&empty);
        zmq_msg_recv (worker, &empty, 0);

        // 获取请求，发送应答
        zmq_msg_t payload;
        zmq_msg_init (&payload);
        zmq_msg_recv (worker, &payload, 0);

        int char_nbr;
        printf ("Worker:");
        for (char_nbr = 0; char_nbr < zmq_msg_size (&payload); char_nbr++)
            printf ("%c", *(char *) (zmq_msg_data (&payload) + char_nbr));
        printf ("\n");

        zmq_msg_init_size (&payload, 2);
        memcpy (zmq_msg_data (&payload), "OK", 2);
```

```
        zmq_msg_send (worker, &address, ZMQ_SNDMORE);
        zmq_close (&address);
        zmq_msg_send (worker, &empty, ZMQ_SNDMORE);
        zmq_close (&empty);
        zmq_msg_send (worker, &payload, 0);
        zmq_close (&payload);
    }
```

而当代码太长时，它不但无法快速地编写，而且也难以理解。到现在为止，我们一直坚持在原生 API 上编程，因为作为 ØMQ 用户，我们需要对它非常了解。但是，当它阻碍我们的工作时，我们必须把它当作一个问题来解决。

我不建议更改 ØMQ API，它是一个在文档中记录的，被成千上万的人认同和依赖的公共合同。我的建议是在这之上构建更高级别的 API。构建依据是我们迄今为止的经验，更具体地说，是我们从编写更复杂的请求 - 应答模式中获得的经验。

我们要的是一个 API，它可以让我们一次性接收和发送整个消息，包括带有任意数量的 104
应答地址的应答封包——一个可以让我们用绝对最少的代码行做我们希望完成的工作的 API。

编写优秀的消息 API 是相当困难的。有一个术语问题：ØMQ 同时使用“消息”来形容多部分消息和单独消息帧。有一个期望值的问题：有时看到消息内容为可打印字符串数据是很自然的，有时为二进制大对象。我们有技术上的挑战，特别是如果我们要避免复制太多外部数据时。

编写一个好的 API 的挑战会影响所有的语言，虽然我的具体用例是 C。但无论使用什么语言，你都要想想如何能有助于你的语言绑定程序，使其与我要描述的 C 语言绑定程序一样好（或更好）。

高级别 API 的特点

我的解决方案是使用三个相当自然而明显的概念：字符串辅助器（已经成为 s_send() 和 s_recv() 的基础）、帧（一个消息帧）和消息（一帧或多帧的列表）。下面是使用这些概念的 API 重写的工人代码：

```
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        zframe_reset (zmsg_last (msg),"OK", 2);
        zmsg_send (&msg, worker);
    }
```

削减我们读取和写入复杂的消息所需的代码量是很棒的：结果易于阅读和理解。让我们

继续该过程来研究使用 ØMQ 工作的其他方面。下面是根据我迄今为止的 ØMQ 经验，想要在一个高级别的 API 中具有的功能的愿望清单。

- **自动处理套接字**。我觉得必须手动关闭套接字，并在一些（但不是全部）情况下必须明确定义 linger 超时，这样很麻烦，如果有一种方法，能在关闭上下文时自动关闭套接字，这将会是很棒的。
- **可移植的线程管理**。每个不简单的 ØMQ 应用程序都使用线程，但 POSIX 线程是不可移植的。一个体面的高级 API 应该将这隐藏在一个可移植层之下。
- **可移植的时钟**。甚至让时间精确到毫秒的分辨率，或休眠几毫秒，这些功能都是不可移植的。现实 ØMQ 应用程序需要可移植的时钟，所以我们的 API 应该提供它们。

105
- **取代 zmq_poll() 的反应器**。轮询简单，但笨拙。编写了很多这类的代码，我们最终重复做同样的工作：计算定时器，在套接字准备就绪时调用代码。一个简单的带有套接字阅读器和计时器的反应器会节省大量的重复工作。
- **正确处理 Ctrl-C**。我们已经看到了如何捕获中断。如果在所有的应用程序中这都会发生将是有益的。

CZMQ 高级别 API

把这个愿望清单在 C 语言中变成现实给了我们 CZMQ（*http://zero.mq/c*），一个 ØMQ 的 C 语言绑定。实际上，这种高级别的绑定是根据本书的早期版本开发出来的。它结合了更好的语义，以便使用一些可移植性层配合 ØMQ 开展工作，以及（对 C 是重要的，但对其他语言不太重要）容器，如散列和列表。CZMQ 也使用了优雅的对象模型，这产生了直白可爱的代码。

示例 3-10 显示了使用更高级别的 API（在 C 中，是 CZMQ）改写的负载均衡代理。

示例3-10：使用高级别API的负载均衡代理（lbbroker2.c）

```
//
//  负载均衡代理
//  演示 CZMQ API 的使用
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      //  工人已经准备就绪的信号

//  使用 REQ 套接字的基本请求 - 应答客户端
//
static void *
```

```
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    //  发送请求，获取应答
    while (true) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  工人使用 REQ 套接字来做负载均衡
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  告诉代理我们已准备好执行工作
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  处理到达的消息
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

106

下面我们来看主任务（参见示例 3-11）。它的功能与以前的 *lbbroker* 例子相同，但它使

用 CZMQ 来启动子线程、维护工人的清单，以及读取和发送消息。

示例3-11：使用高级API的负载均衡代理（lbbroker2.c）：主任务

```
int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "ipc://frontend.ipc");
    zsocket_bind (backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  可用的工人的队列
    zlist_t *workers = zlist_new ();
```

负载均衡器主循环如示例 3-12 所示。它的工作方式与前面的示例相同，但代码短了很多，这是因为 CZMQ 给了我们一个利用更少的调用做更多工作的 API。

107 示例3-12：使用高级API的负载均衡代理（lbbroker2.c）：负载均衡器主循环

```
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  只在我们有可用的工人时才轮询前端
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  中断

        //  在后台处理工人活动
        if (items [0].revents & ZMQ_POLLIN) {
            //  使用工人身份以用于负载均衡
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  中断
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);

            //  转发消息给客户端，如果它不是一个 READY 消息
            zframe_t *frame = zmsg_first (msg);
```

```
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  获取客户端请求，路由到第一个可用的工人
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  当我们完成后，执行适当的清理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

CZMQ 提供了彻底的中断处理功能。这意味着，Ctrl-C 将导致任何阻塞的 ØMQ 调用以返回代码 -1 退出，并将 errno 设置为 EINTR。在这种情况下，高级的 recv 方法将返回 NULL。所以，你可以像这样完全地退出一个循环：

108

```
while (true) {
    zstr_send (client, "HELLO");
    char *reply = zstr_recv (client);
    if (!reply)
        break;              //  中断
    printf ("Client: %s\n", reply);
    free (reply);
    sleep (1);
}
```

或者，如果你调用 zmq_poll()，那么可以这样测试返回代码：

```
if (zmq_poll (items, 2, 1000 * 1000) == -1)
    break;              //  中断
```

前一个例子仍然使用 zmq_poll()。那么，怎么使用反应器呢？CZMQ zloop 反应器很简单，但功能强大。它可以让你：

- 在任何套接字上设置一个阅读器（即当套接字具有输入时要调用的代码）。
- 取消套接字上的阅读器。
- 设置以特定的间隔一次或多次触发的定时器。
- 取消计时器。

当然，zloop 在内部使用 zmq_poll()。每次添加或移除阅读器时，它都会重建其轮询集合，并且计算轮询超时以匹配下一个定时器。然后，它对需要注意的每个套接字和定时器调用读取器和定时器处理程序。

当我们使用一个反应器的模式时，我们的代码翻转了过来。主要的逻辑看上去是这样的：

```
zloop_t *reactor = zloop_new ();
zloop_reader (reactor, self->backend, s_handle_backend, );self
zloop_start (reactor);
zloop_destroy (&reactor);
```

消息的实际处理位于专用函数或方法内部。你可能不喜欢这种风格，但这是一个品味的问题。它所做的帮助是混合定时器和套接字活动。在本文的其余部分，在简单的例子中，我们将使用 zmq_poll()，而在更复杂的例子中使用 zloop。

示例 3-13 显示了再次重写的负载均衡代理，这一次使用 zloop。

示例3-13：使用zloop的负载均衡代理（lbbroker3.c）

```
//
//  负载均衡代理
//  演示 CZMQ API 的使用和反应器样式
//
//  客户端和工人任务与前一个例子完全相同
...
//  负载均衡结构，被传递给反应器处理程序
109 typedef struct {
    void *frontend;             //  监听客户端
    void *backend;              //  监听工人
    zlist_t *workers;           //  准备就绪的工人的列表
} lbbroker_t;
```

在反应器的设计中，每当一个消息到达套接字时，反应器就将其传递给一个处理函数。我们有两个处理函数，一个用于前端，另一个用于后端，并如示例 3-14 所示。

示例3-14：使用zloop的负载均衡代理（lbbroker3.c）：反应器的设计

```
//  在前台处理来自客户端的输入
int s_handle_frontend (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
```

```
    lbbroker_t *self = (lbbroker_t *) arg;
    zmsg_t *msg = zmsg_recv (self->frontend);
    if (msg) {
        zmsg_wrap (msg, (zframe_t *) zlist_pop (self->workers));
        zmsg_send (&msg, self->backend);

        //  如果我们从 1 个工人变成 0 个工人，则取消在前端的阅读器
        if (zlist_size (self->workers) == 0) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
            zloop_poller_end (loop, &poller);
        }
    }
    return 0;
}

//  在后端处理来自工人的输入
int s_handle_backend (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    //  使用工人身份以用于负载均衡
    lbbroker_t *self = (lbbroker_t *) arg;
    zmsg_t *msg = zmsg_recv (self->backend);
    if (msg) {
        zframe_t *identity = zmsg_unwrap (msg);
        zlist_append (self->workers, identity);

        //  如果我们从 0 个工人变成 1 个工人，就在前端启用阅读器
        if (zlist_size (self->workers) == 1) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
            zloop_poller (loop, &poller, s_handle_frontend, self);
        }
        //  转发消息给客户端，如果它不是一个 READY 消息
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
            zmsg_destroy (&msg);
        else
            zmsg_send (&msg, self->frontend);
    }
    return 0;
}
```

110

主任务（参见示例 3-15）现在设置子任务，然后启动它的反应器。如果你按 Ctrl-C，反应器退出且主任务关闭。因为该反应器是一个 CZMQ 类，这个例子可能不能同样好地翻译成所有语言。

示例3-15：使用zloop的负载均衡代理（lbbroker3.c）：主任务

```
int main (void)
{
    zctx_t *ctx = zctx_new ();
    lbbroker_t *self = (lbbroker_t *) zmalloc (sizeof (lbbroker_t));
    self->frontend = zsocket_new (ctx, ZMQ_ROUTER);
    self->backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (self->frontend, "ipc://frontend.ipc");
    zsocket_bind (self->backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  可用的工人的队列
    self->workers = zlist_new ();

    //  准备反应器并发射它
    zloop_t *reactor = zloop_new ();
    zmq_pollitem_t poller = { self->backend, 0, ZMQ_POLLIN };
    zloop_poller (reactor, &poller, s_handle_backend, self);
    zloop_start  (reactor);
    zloop_destroy (&reactor);

    //  当我们完成后，执行适当的清理
    while (zlist_size (self->workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (self->workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&self->workers);
    zctx_destroy (&ctx);
    free (self);
    return 0;
}
```

若要实现当你给应用程序发送 Ctrl-C 时，它们能够正常关闭，这可能会非常棘手。如果你使用 zctx 类，它会自动设置信号处理，但你的代码还必须配合。你必须跳出任何循环，如果 zmq_poll() 返回 -1 或者 zstr_recv()、zframe_recv() 或 zmsg_recv() 方法中任何一个返回 NULL。如果你有嵌套循环，使得外循环的条件是 !zctx_interrupted 可能是有用的。

异步客户端/服务器模式 111

在 ROUTER 到 DEALER 的例子中，我们看到了一个 1 对 N 的用例，其中一个服务器异步地与多个工人交流。我们可以把这个颠倒过来，从而得到一个非常有用的 N 对 1 的架构，其中不同的客户端与一台服务器交流，而且这是异步执行的（参见图 3-12）。

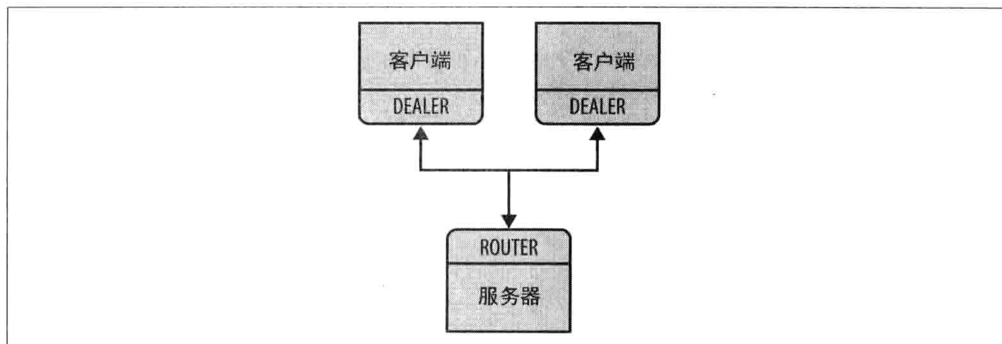


图3-12：异步客户端/服务器

下面是它的工作原理：

- 客户端连接到服务器并发送请求。
- 对于每一个请求，服务器发送 0 个或多个应答。
- 客户端可以发送多个请求，而无须等待应答。
- 服务器可以发送多个应答，而无须等待新的请求。

示例 3-16 显示了它的实现方法。

示例3-16：异步客户端/服务器（asyncsrv.c）

```
//
//  异步客户端到服务器（DEALER 到 ROUTER）
//
//  虽然本示例在一个进程中运行，但这只是为了
//  易于启动和停止这个示例。每个任务都有其
//  自己的上下文并从概念上作为一个独立的进程。

#include "czmq.h"
//  ---------------------------------------------------------------------   112
//  这是我们的客户端任务。
//  它连接到服务器，然后每秒钟发送一个请求。
//  它收集到达的响应，并将它们输出出来。
//  我们将并行运行多个客户端，每个客户端都有不同的随机 ID。
```

```
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_DEALER);

    //  设置随机的身份以便跟踪
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    zsockopt_set_identity (client, identity);
    zsocket_connect (client, "tcp://localhost:5570");

    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    int request_nbr = 0;
    while (true) {
        //  每秒钟嘀嗒一次，提取到达的消息
        int centitick;
        for (centitick = 0; centitick < 100; centitick++) {
            zmq_poll (items, 1, 10 * ZMQ_POLL_MSEC);
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (client);
                zframe_print (zmsg_last (msg), identity);
                zmsg_destroy (&msg);
            }
        }
        zstr_sendf (client, "request #%d", ++request_nbr);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

我们的服务器任务在示例 3-17 中给出。它采用了多线程服务器模型来处理来自工人池的请求并将应答路由回客户端。一个工人一次只可以处理一个请求，而一个客户端可以同时与多个工人交流。

示例3-17：异步客户端/服务器（asyncsrv.c）：服务器任务

```
static void server_worker (void *args, zctx_t *ctx, void *pipe);

void *server_task (void *args)
{
    zctx_t *ctx = zctx_new ();


    //  前端套接字通过 TCP 与客户端交流
```

113

```
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5570");


    // 后端套接字通过 inproc 与工人交流
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://backend");

    // 启动工人线程池，准确数字是不重要的
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++)
        zthread_fork (ctx, server_worker, NULL);

    // 后端通过一台代理服务器连接到前端
    zmq_proxy (frontend, backend, NULL);

    zctx_destroy (&ctx);
    return NULL;
}
```

每个工人任务同一时间在一个请求上工作，并且发送回随机数量的应答，应答之间有随机的延迟，如示例 3-18 所示。

示例3-18：异步客户端/服务器（asyncsrv.c）：工人任务

```
static void
server_worker (void *args, zctx_t *ctx, void *pipe)
{
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "inproc://backend");

    while (true) {
        // DEALER 套接字给我们应答封包和消息
        zmsg_t *msg = zmsg_recv (worker);
        zframe_t *identity = zmsg_pop (msg);
        zframe_t *content = zmsg_pop (msg);
        assert (content);
        zmsg_destroy (&msg);

        // 发回 0..4 应答
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            // 休眠一秒钟的一部分
            zclock_sleep (randof (1000) + 1);
            zframe_send (&identity, worker, ZFRAME_REUSE + ZFRAME_MORE);
            zframe_send (&content, worker, ZFRAME_REUSE);
```

```
        }
        zframe_destroy (&identity);
        zframe_destroy (&content);
    }
}
```

114

```
//  主线程只是启动几个客户端和一个服务器，
//  然后等待服务器完成。

int main (void)
{
    zthread_new (client_task, NULL);
    zthread_new (client_task, NULL);
    zthread_new (client_task, NULL);
    zthread_new (server_task, NULL);

    //  运行 5 秒钟，然后退出
    zclock_sleep (5 * 1000);
    return 0;
}
```

该示例在一个进程中运行，使用多线程模拟一个真正的多进程架构。当运行这个例子时，你会看到三个客户端（每个客户端都带有一个随机 ID），输出它们从服务器获取的应答。仔细观察，你会看到每个客户端任务都会得到每个请求的零个或多个应答。

有关这段代码的一些注释：

- 客户端每秒发送一次请求，并取回零个或多个应答。要使用 zmq_poll() 完成这项工作，我们不能简单地用 1 秒钟的超时时间来轮询，否则最终仅会在我们收到最后的应答一秒钟后才发送一个新的请求。因此，我们以高频率轮询（以每次轮询百分之一秒的速度轮询 100 次），这大约是准确的。
- 服务器使用一个工人线程池，每个线程都同步处理一个请求。它使用一个内部队列将这些工人线程连接到其前端的套接字。它使用一个 zmq_proxy() 调用来连接前端和后端的套接字。

图 3-13 显示了本示例架构的详细视图。请注意，我们正在客户端和服务器之间做 DEALER 到 ROUTER 的对话，但在内部，在服务器的主线程和工人之间，我们正在做 DEALER 到 DEALER 的对话。如果工人是严格同步的，我们会使用 REP，但由于我们要发送多个应答，因此需要一个异步套接字。我们不希望路由应答，它们总是去往那个发送给我们请求的单个服务器线程。

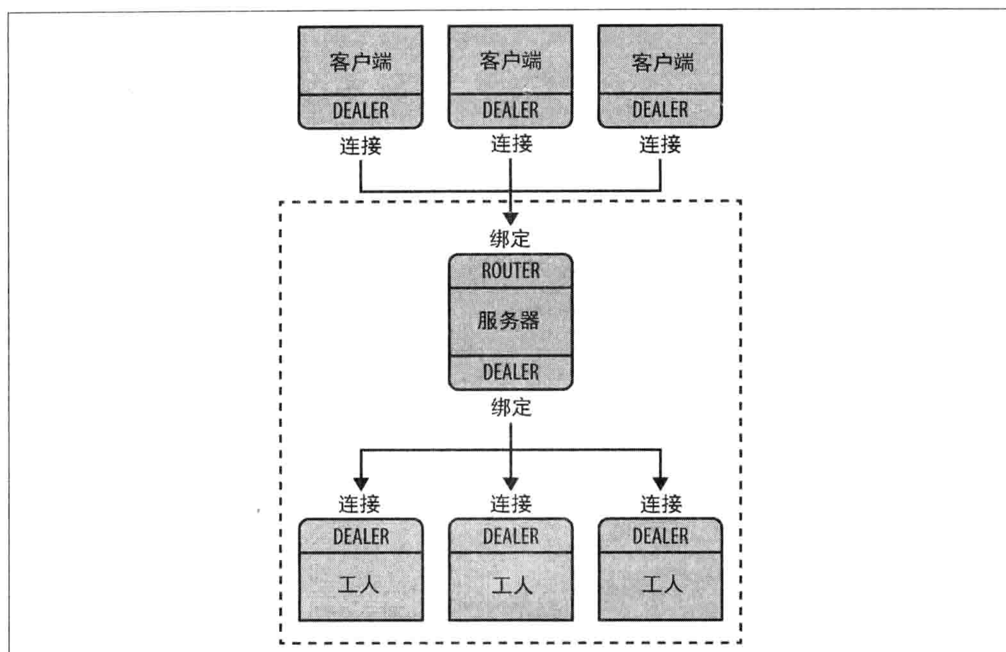


图3-13：异步服务器的详情

让我们来考虑路由封包。客户端发送一个简单的消息，服务器线程接收一个两部分消息（真正的消息加上客户端的身份前缀）。我们将这个发送给工人，工人把它作为一个正常的应答封包，并返回给我们。然后，我们可以用它来把应答路由回正确的客户端： 115

```
     客户端          服务器           前端           工人
[ DEALER ]<---->[ ROUTER <----> DEALER <----> DEALER ]
            1 部分          2 部分          2 部分
```

现在对于套接字：我们可以使用负载均衡的 ROUTER 到 DEALER 模式去跟工人交流，但它需要额外的工作。在这种情况下，DEALER 到 DEALER 模式可能是不错的：权衡是对每个请求有更低的延迟，但工作不均衡分布的风险更高。在这种情况下，简单胜出。

当你构建与客户端保持有状态会话的服务器时，会遇到一个经典的问题。如果服务器保留每个客户端的某个状态，而客户端不断地进进出出，那么最终服务器将耗尽资源。即使是同一个客户端保持连接，如果你使用的是默认的身份，那么每个连接看起来都会像一个新连接。

在这个例子中，我们通过只把状态保留很短的时间（工人处理一个请求所需的时间），然后扔掉该状态来欺骗。但是，对于很多情况，这是不实际的。要在一个有状态的异步服务器中妥善管理客户端的状态，你必须做到以下几点：

116 • 执行从客户端到服务器的信号检测。在我们的例子中，每秒一次地发送一个请求，它可以可靠地用作信号检测。
• 使用客户端的身份（无论是生成的或显式的）作为键来存储状态。
• 检测一个停止的检测信号。比如说，如果两秒内没有从某个客户端发出来请求，服务器就可以检测到这一点，并销毁它所持有的该客户端的任何状态。

能够工作的示例：跨代理路由

让我们把到目前为止看到的一切东西扩展为一个真正的应用程序。我们将在几个迭代中逐步构建这个应用程序。

假设我们最好的客户急切地打电话给我们，要求提供一个大型云计算设施的设计。他有这样一个愿景，其中有跨越许多数据中心，并且作为一个整体来工作的一个云，而每个数据中心都是客户端和工人的集群。

因为我们足够聪明，知道实践总是胜过理论，我们建议使用 ØMQ 做一个能工作的模拟。我们的客户，急于在他自己的老板改变主意前锁定预算，并已在 Twitter 上阅读了 ØMQ 的功绩，对此表示赞同。

建立详情

几杯浓缩咖啡之后，我们希望跳进编写代码的任务中，但是有一个小小的声音告诉我们，在对完全错误的问题做一个耸人听闻的解决方案之前，需要获得更多的详情。“这个云在做什么样的工作呢？”我们问。客户解释道：

• 工人在各种硬件上运行，但它们都能够处理任何任务。每个集群有几百个工人，并且总共有多达十几个集群。
• 客户端为工人创建任务。每个任务是工作的一个独立单位，所有的客户端都希望找到一个可用的工人，并尽快向它发送任务。将会有很多客户端，它们会随意地来来去去。
• 真正困难的是要能随时添加和删除集群。一个集群可以瞬间离开或加入云，它带着其所有的工人和客户端。
• 如果在自己的集群中没有工人，客户端的任务将跑到在云中其他可用的工人那里。
• 客户端在一个时间发送一个任务，并等待应答。如果在 X 秒之内没有得到应答，它们就会再次发送任务。这不是我们所关注的，客户端 API 已经做了这项工作。
117 • 工人在一个时间处理一个任务，它们是非常简单的野兽。如果它们崩溃了，它们会被任何一个启动它们的脚本重新启动。

所以我们会仔细检查，以确保我们正确地理解了这些要点：

- “集群之间将会有某种形式的超级网络互联，对不对？”我们问。客户说，“是的，当然，我们不是白痴。”
- “我们在谈论的数量是什么样的？”我们问。客户回答说：“每个集群最多 1000 个客户端，每个客户端每秒最多处理 10 个请求。请求很少，应答也很少，它们每个都不超过 1KB 字节。”

所以我们做一个小小的计算，并看到这将在普通的 TCP 上很好地工作。2500 客户端 ×10/s×1000 字节 ×2 方向 =50MB/s 或 400Mb/s，千兆网络处理这个任务是没有问题的。

这是一个简单的问题，它不需要特别的硬件或协议，而只要一些聪明的路由算法和精心的设计。我们从设计一个集群（一个数据中心）开始，然后再弄清楚如何把集群连接在一起。

单集群架构

工人和客户端是同步的。我们想要使用负载均衡模式路由任务给工人。工人都是相同的，我们的设施对不同的服务没有概念。工人是匿名的，客户端不能直接处理它们。我们在这里没有尝试提供有保证的传递、重试，等等。

因为我们已经介绍过的原因，客户端和工人不会彼此直接对话，这使得无法动态地添加或删除节点。因此，我们的基本模型包括前面看到的（参见图 3-14）请求 - 应答消息代理。

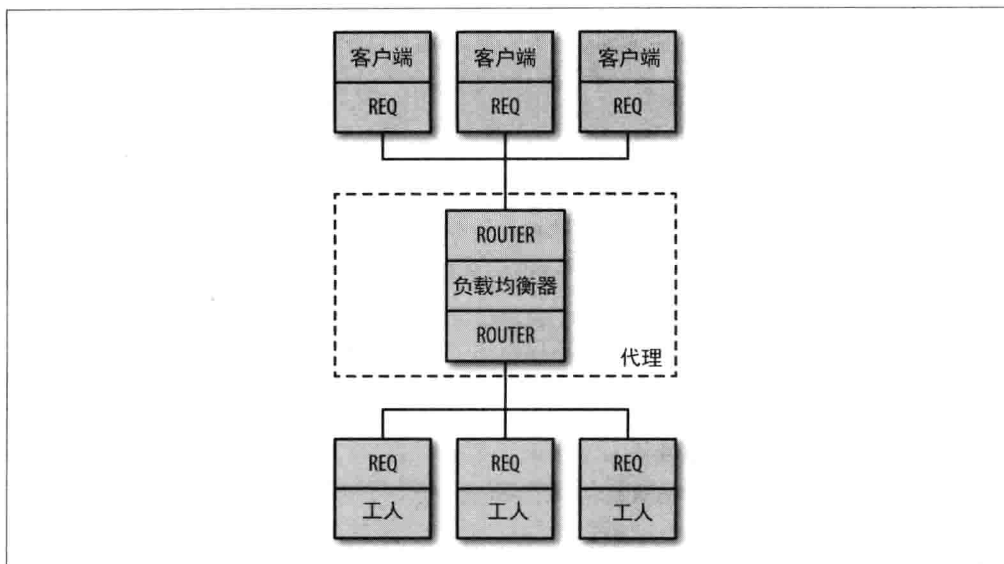


图3-14：集群架构

118

扩展到多个集群

现在我们将这扩展到多个集群。每个集群都有一组客户端和工人，以及将这些连接在一起的一个代理，如图 3-15 所示。

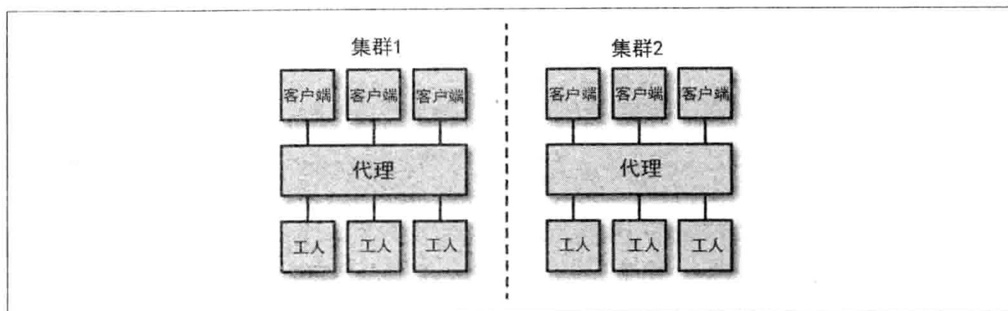


图3-15：多个集群

现在的问题是，如何让每个集群的客户端与其他集群的工人交流呢？这里有几个可能的办法，它们各有利弊：

- 客户端可以直接同时连接到两个代理。好处是，我们并不需要修改代理或工人。然而，客户端变得越来越复杂，并且要能感知到总体的拓扑结构。例如，如果我们想增加
119 第三个或第四个集群，客户端就都会受到影响。实际上，我们必须将路由和故障转移逻辑移动到客户端，而那是不好的。
- 工人可能直接连接到两个代理。但 REQ 工人不能做到这一点，它们只能应答一个代理。我们可能会使用 REP，但 REP 不允许我们定制代理到工人的路由，就像负载均衡所做的那样，而只有内置的负载均衡。这是一个失败，如果我们想将工作分配给空闲的工人，恰恰需要负载均衡。一种解决方案是让工作节点使用 ROUTER 套接字，让我们将这个方法标记为“思路＃ 1”。
- 代理可以彼此连接。这看起来是最巧妙的，因为它创建了最少的附加连接。我们不能动态添加集群，但是不管怎样，这可能超出了我们的范围。在这个解决方案中，客户端和工人对真实的网络拓扑保持无知，而代理在它们有富余处理能力时会将这个情况告诉对方。让我们把这个方法标记为“思路＃ 2”。

下面让我们来探讨思路＃ 1。在这个模型中，我们拥有同时连接到两个代理并从任何一个代理接受作业的工人（参见图 3-16）。

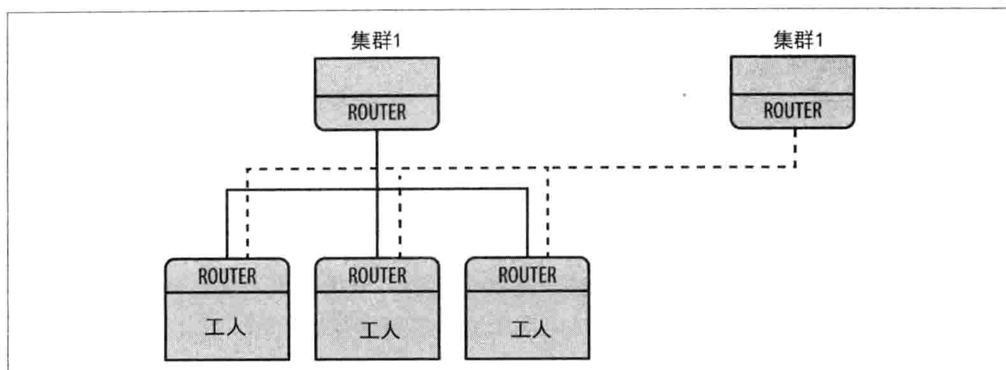


图3-16：思路1：交叉连接的工人

它看起来是可行的。然而，它并没有提供我们想要的东西，在这种方法中，客户端只要有可能就会得到本地工人，而只有当这比等待更好时，才会得到远程工人。此外，工人将对两个代理同时显示“准备就绪”，所以可能一次得到两份工作，而其他工人仍保持空闲状态。看来这样的设计是失败的，因为我们再次把路由逻辑推到了边缘。

因此，再来看思路＃ 2。我们将代理互联起来并且不接触客户端或工人，这就像我们已经习惯了的 REQ（参见图 3-17）。

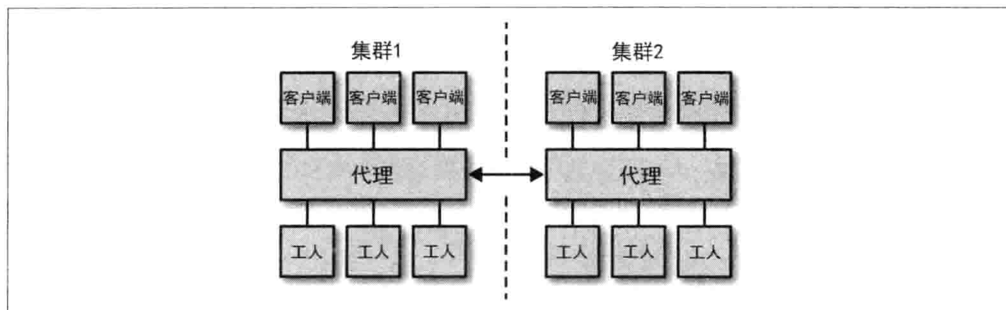


图3-17：思路2：代理互相交流

这种设计是有吸引力的，因为问题在一个地方解决，其他地方对此是看不见的。基本上， 120
代理打开秘密通道，双方像骆驼商人那样说悄悄话，“嘿，我有一些富余处理能力。如果你有太多的客户端请给我留言，我们会处理。”

实际上，这仅仅是一个更复杂的路由算法：代理成为双方的分包商。这样的设计还有其他令人喜欢的东西，甚至在我们演练真正的代码之前：

- 它把常见的情况（相同的集群上的客户端和工人）作为默认设置，并且为特殊情况（在集群之间混合作业）做一些额外工作。
- 它让我们对不同类型的工作使用不同的消息流。这意味着我们可以用不同的方式处理它们，例如使用不同类型的网络连接。
- 感觉它会顺利扩展。相互连接三个或更多的代理并没有变得过于复杂。如果我们觉得这是一个问题，则很容易通过添加一个超级代理来解决。

现在，我们将做一个实例。我们会将整个集群包装到一个进程中。这显然是不现实的，但它可以很方便地模拟，而这种模拟可以准确地扩展到实际的进程。这正是 ØMQ 的美好之处：你可以在微观层次设计并将它扩展到宏观层面。线程可以变成进程，然后变成电脑，而模式和逻辑保持相同。我们的每一个“集群”进程中都包含客户端线程、工作线程和代理线程。

我们现在对基本模式非常了解了：

- REQ 客户端（REQ）线程创建工作负载，并将它们传递到代理（ROUTER）。
- REQ 工人（REQ）线程处理工作负载，并将结果返回到代理（ROUTER）。
- 121 代理使用的负载均衡模式对工作负载进行排队和分配。

联盟与对等比较

互联代理有几种可能的方式。我们需要的是能够告诉其他代理，“我们有处理能力”，然后接收多个任务。我们还需要能够告诉其他代理，“停，我们满载了。”它不需要是完美的，有时我们可能会接受我们不能立即处理的工作，但会尽快完成它们。

最简单的互联是联盟（*federation*），其中代理分别为客户端和工人模拟对方。我们通过将我们的前端连接到其他代理的后端套接字来做到这一点（参见图 3-18）。请注意，将套接字既绑定到一个端点，又将其连接到其他端点，这是合法的。

这将使我们的两个代理都有简单的逻辑和相当好的机制：当没有客户端时，就告诉另一个代理“就绪”，并从它接受一项工作。但问题是，它对这一问题显得过于简单。一个联盟代理在同一个时间将只能处理一个任务。如果代理是模拟步调一致的客户端和工人，理论上它也将是步调一致的，并且如果它有很多可用的工人，它们也不会被使用。我们的代理需要以完全异步的方式进行连接。

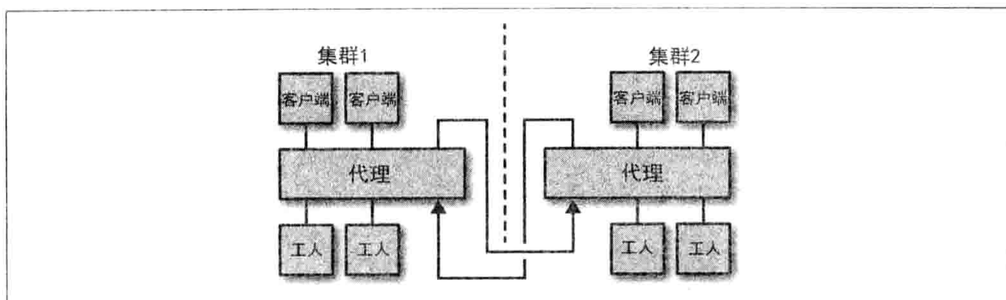


图3-18：在联盟模型中交叉连接的代理

对于其他种类的路由，联盟模型是完美的，尤其是面向服务的架构（SOA），它根据服务名称和接近度，而不是负载均衡或轮询来路由。所以不要把它作为无用的模型排除掉，它只是不适合所有用例而已。

与联盟相反，让我们来看一个对等的方法，其中每个代理都明确了解对方，并通过特权通道交流。让我们分析一下这种情况，假设我们要互联 *N* 个代理。每个代理有（*N*-1）个对等节点，而所有代理都使用完全相同的代码和逻辑。在代理间存在两种不同的信息流：

- 每个代理都需要随时告诉它的对等节点它有多少工人可用。这可以是相当简单的信息，只是定期更新的一个量。对于这个，明显（并且正确）的套接字模式是发布 - 订阅：每个代理都打开 PUB 套接字，并在它上面公布状态信息，并且每一个代理也都打开一个 SUB 套接字，且将它连接到其他所有代理的 PUB 套接字上，以得到来自对等节点的状态信息。 122
- 每个代理都需要有一种方法来将任务委托给对等节点并将应答异步地取回来。我们将使用 ROUTER 套接字做到这一点，其他的组合都不会起作用。每个代理都有两个这样的套接字：一个用于它接收的任务，另一个用于它委托的任务。如果我们不使用两个套接字，每次要知道我们在读的东西是请求还是应答，这将需要更多的工作，这将意味着往消息封包添加更多的信息。

而且还有一个代理及其本地客户和工人之间的信息流。

命名规范

3 个流 × 每个流 2 个套接字 =6 个，这是我们必须在代理中管理的套接字。选择好名字对于一个多套接字杂耍行为在我们的脑海中保持合理的连贯至关重要。套接字做某些工作，而它们应该根据它们的名字来做工作。这与编写可读性良好的代码一样，这使你能够在一个寒冷的周一上午，在喝咖啡前毫不痛苦地阅读几个星期前写的代码。

让我们为套接字执行萨满教的命名仪式。这三个流分别是：

- 代理及其客户端和工人之间的本地（*local*）请求 - 应答流。
- 代理及其对等代理之间的云端（*cloud*）请求 - 应答流。
- 代理及其对等代理之间的状态（*state*）流。

寻找长度相同且有意义的名称使得我们的代码将能很好地对齐。这不是一件大事，但对细节的关注是有帮助的。对于每一个流，代理都有两个套接字，我们可以互不相关地称之为前端和后端。我们经常使用这些名字。前端接收信息或任务，后端发送那些信息或任务给其他节点。概念上流是从前面到后面的（而应答按相反的方向运动，从后到前）。

因此，在为这个教程写的所有代码中，我们将利用下面这些套接字名称：

- `localfe` 和 `localbe` 用于本地流
- `cloudfe` 和 `cloudbe` 用于云端流
- `statefe` 和 `statebe` 用于状态流

123 对于我们的传输，因为是在一台电脑中模拟整个事情，所以将全部使用 `ipc`。在连通性方面，也有类似 `tcp` 的工作优势（即，它是一个断开连接的传输，不像 `inproc`），但我们并不需要 IP 地址或 DNS 名称，在这里它们将是痛苦的。相反，我们将使用称作 `something-local`、`something-cloud` 和 `something-state` 的 `ipc` 端点，这里的 `something` 是我们的模拟集群的名称。

你可能会想，对一些名字，这是大量的工作。为什么不把它们称作 `s1`、`s2`、`s3`、`s4`，等等？答案是，如果你的大脑不是一部完美的机器，你阅读代码时就需要大量的帮助，我们将会看到，这些名字是有帮助的。“3 个流，2 个方向”比“6 种不同的套接字”更容易记住（参见图 3-19）。

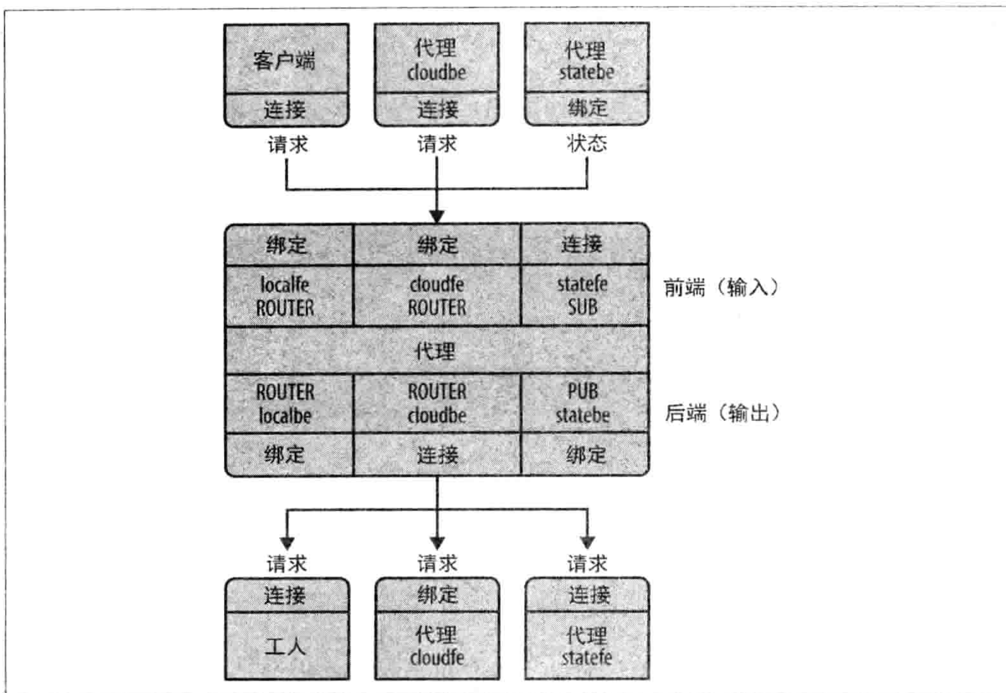


图3-19：安排代理套接字

请注意，我们将每个代理中的`cloudbe`连接到其他所有代理中的`cloudfe`，同样，我们将每个代理中的`statebe`连接到其他所有代理中的`statefe`。

状态流原型

因为每种套接字流对于粗心大意的人都有它自己的小陷阱，所以我们将在真实的代码中逐个测试它们，而不是试图一气呵成地将它们抛在一大堆代码中。当我们对每个流都很熟练后，就可以把它们组合成一个完整的程序，我们将从状态流开始（参见图 3-20）。 124

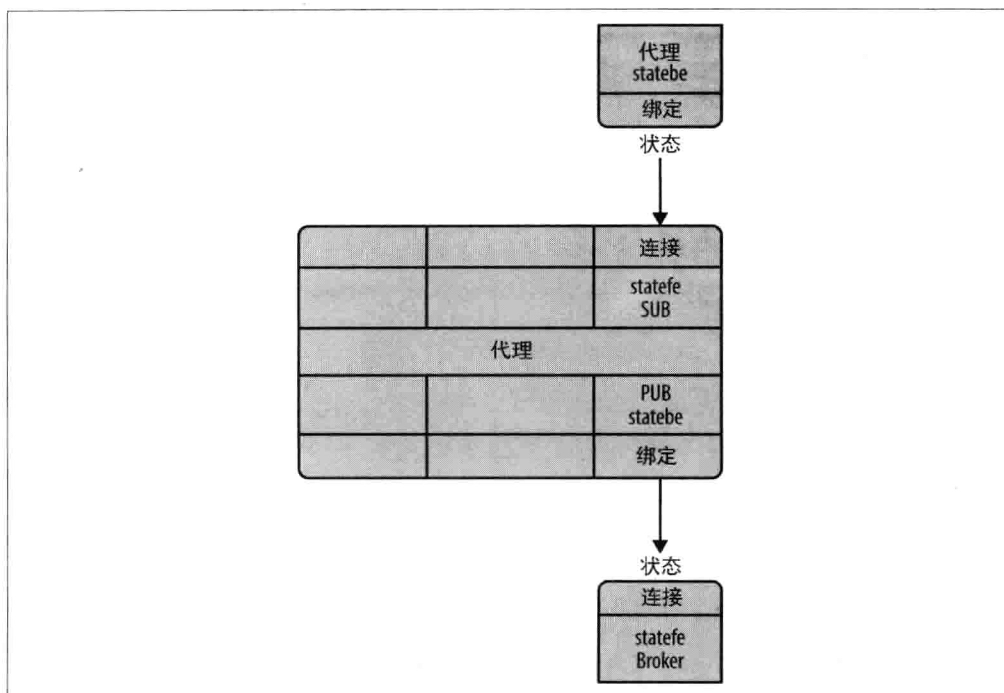


图3-20：状态流

示例 3-19 显示了这是如何在代码中实现的。

示例3-19：状态流原型（peering1.c）

```
//
//  代理对等节点模拟（第 1 部分）
//  状态流的原型
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    //  第一个参数是此代理的名称
    //  其他参数是我们的对等节点的名称
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    char *self = argv [1];
125    printf ("I: preparing broker at %s...\n", self);
```

```
    srandom ((unsigned) time (NULL));

    zctx_t *ctx = zctx_new ();

    //  把状态后端绑定到端点
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  把 statefe 连接到所有对等节点
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    zsockopt_set_subscribe (statefe, "");
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
```

主循环（参见示例 3-20）将状态消息发送到对等节点并收集从这些节点回来的状态信息。zmq_poll() 超时定义了我们自己的信号检测节奏。

示例3-20：状态流原型（peering1.c）：主循环

```
    while (true) {
        //  轮询活动，或 1 秒超时时间
        zmq_pollitem_t items [] = { { statefe, 0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        //  处理传入的状态消息
        if (items [0].revents & ZMQ_POLLIN) {
            char *peer_name = zstr_recv (statefe);
            char *available = zstr_recv (statefe);
            printf ("%s - %s workers free\n", peer_name, available);
            free (peer_name);
            free (available);
        }
        else {
            //  为工人的可用性发送随机值
            zstr_sendm (statebe, self);
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

关于这段代码的注释：

126
- 每个代理都有一个我们用来构建 ipc 端点名称的标识。一个真正的代理则需要使用 TCP 和更复杂的配置方案。我们将在本书的后面查看这样的方案，但就目前而言，使用生成的 ipc 名字让我们忽略了从哪里得到 TCP/IP 地址或名称的问题。
- 我们使用一个 zmq_poll() 循环作为程序的核心。它处理传入的消息，并发送出状态消息。只有当我们没有得到任何传入的消息并且等了 1 秒后，才发送状态消息。如果我们每次在得到一个传入消息时就发送一个状态消息，会造成消息风暴。
- 我们使用一个由发送者的地址和数据组成的两部分的发布 - 订阅消息。需要注意的是，我们需要知道发布者的地址，以便向其发送任务，而明确发送它的唯一办法就是将其作为消息的一部分发送。
- 我们不设置订阅者身份，因为如果这样做的话，在连接到运行的代理时就会得到过时的状态信息。
- 我们不会在发布者上设置 HWM，但如果使用 ØMQ v2.x 版本，这将是一个明智的想法。

我们可以构建这个小程序并通过运行它三次来模拟三个集群。让我们分别称它们为 DC1、DC2 和 DC3（名字是任意的）。运行下面这三个命令，每个命令都在单独的窗口中运行：

```
peering1 DC1 DC2 DC3  #  启动 DC1，并连接到 DC2 和 DC3
peering1 DC2 DC1 DC3  #  启动 DC2，并连接到 DC1 和 DC3
peering1 DC3 DC1 DC2  #  启动 DC3，并连接到 DC1 和 DC2
```

你会看到每个集群报告其节点的状态，并在几秒后，它们都会兴高采烈地每秒打印一次随机数。试试这个，并用三个代理都与每秒的状态更新匹配和同步来满足自己。

在现实生活中，我们不会每隔一段时间就发送状态消息，而是每当有一个状态变化，即，当一个工人变得可用或不可用时就发送状态消息。这似乎是一个巨大的流量，但状态消息都很小，并且我们已经建立了集群间的超快连接。

如果我们想以精确的时间间隔发送状态信息，需要创建一个子线程，然后在该线程中打开 statebe 套接字。然后，我们会从主线程把不定时的状态更新发送到子线程，并让子线程将它们与常规传出的消息混在一起。但是，这些工作超出了我们在这里需要的范围。

本地流和云端流原型

现在，让我们通过本地和云套接字（参见图 3-21）编写任务流的原型。这段代码从客户端提取请求，然后以随机方式将它们分发到本地的工人和云端的对等节点上。

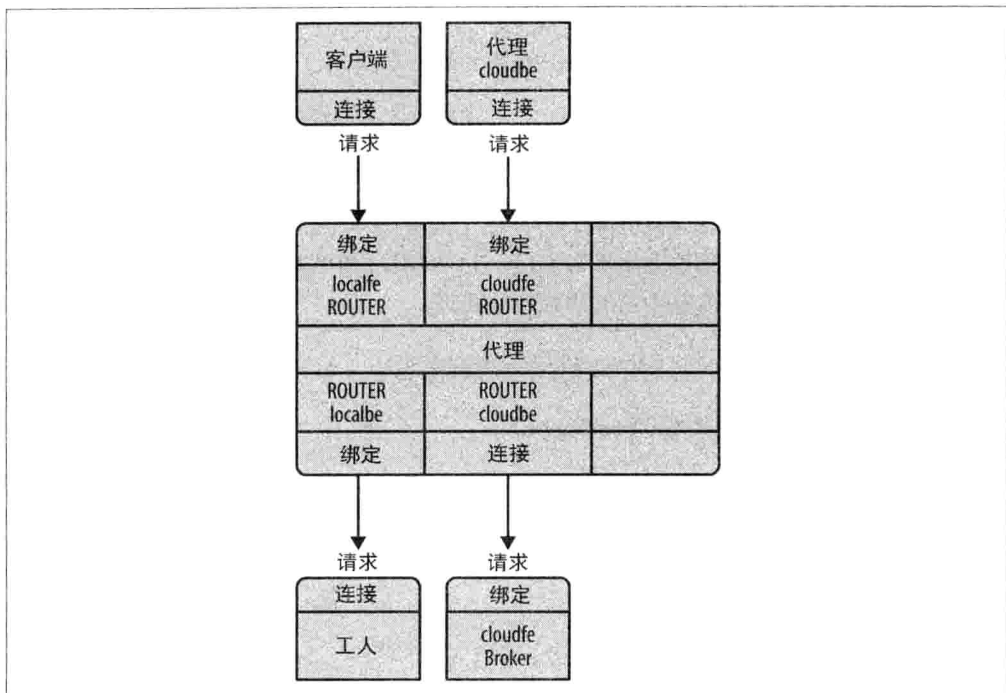


图3-21：任务流

在进入这越来越复杂的代码之前，让我们先来勾勒核心路由逻辑，并把它分解成一个简单但强大的设计。 127

我们需要两个队列，一个用于存放来自本地客户端的请求，一个用于存放来自云客户端的请求。其中一个办法是将消息从本地和云前端提取出来，并把这些消息输送到各自的队列中。但是，这是一项毫无意义的工作，因为 ØMQ 套接字已经是队列了。因此，让我们采用 ØMQ 套接字缓冲区作为队列。

这是我们在本章前面的负载均衡代理中使用过的技术，而它工作得很好。我们仅当有地方可用来发送请求时，才从两个前端读取。我们始终可以从后端读取，因为它们给了我们路由回去的应答。只要后端不是正在和我们交流，甚至查看前端都是没有意义的。

所以，我们的主循环就变成了：

- 轮询后端的活动。当我们得到一个消息时，它可能会是来自一个工人的"准备就绪"，也可能是一个应答。如果它是一个应答，我们就通过本地或云前端将它路由回去。
- 如果工人已经应答了，它就成为可用的，所以我们将它排入队列并对它计数。

128 • 当存在可用的工人时，我们从任一前端获取一个请求（如果有的话）并将它路由到一个本地工人，或随机地路由到云中的某个对等节点。

将任务随机发送给对等的代理，而不是工人，这模拟了整个集群的工作分配。这有点傻，但在这个阶段是没有问题的。

我们使用代理身份在代理之间路由消息。在这个简单的原型中，每个代理都有我们在命令行上提供的名字。只要这些名字不与 ØMQ 生成的用于客户端节点的 UUID 发生重叠，我们就可以确定应当将应答路由给客户端还是代理。

示例 3-21 到示例 3-26 显示了上述思路在代码中的实现。

示例3-21：本地和云端流原型（peering2.c）

```
//
//  代理对等节点模拟（第 2 部分）
//  请求 - 应答流的原型
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      //  工人已准备就绪的信号

//  我们自己的名字，在实践中这将在每个节点上配置
static char *self;
```

客户端的任务，如示例 3-22 所示，使用一个标准的同步 REQ 套接字实现了一个请求 - 应答对话。

示例3-22：本地和云端流原型（peering2.c）：客户端任务

```
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);

    while (true) {
        //  发送请求，获取应答
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              //  中断
        printf ("Client: %s\n", reply);
```

```
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

工人的任务使用 REQ 套接字插入负载均衡器，如示例 3-23 所示。

示例3-23：本地和云端流原型（peering2.c）：工人的任务

```
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  告诉代理我们已准备好执行工作
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  处理到达的消息
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断

        zframe_print (zmsg_last (msg), "Worker:");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```
129

主任务以设立其前端和后端的套接字开始，然后启动它的客户端和工人任务（参见示例 3-24）。

示例3-24：本地和云端流原型（peering2.c）：主任务

```
int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是我们的对等节点名称
    //
    if (argc < 2) {
```

```
        printf ("syntax: peering2 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    zctx_t *ctx = zctx_new ();

130 //  把云前端绑定到端点
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe,"ipc://%s-cloud.ipc", self);

    //  把云后端连接到所有节点
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%s'\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }
    //  准备本地前端和后端
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);
    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  让用户告诉我们何时我们可以启动……
    printf ("Press Enter when all brokers are started: ");
    getchar ();

    //  启动本地工人
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  启动本地客户端
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);
```

接下来，我们处理请求-应答流（参见示例3-25）。我们使用负载均衡在任何时候轮询工人，并仅当存在可用的一个或多个工人时才轮询客户端。

示例3-25：本地和云端流原型（peering2.c）：请求-应答处理

```
// 可用的工人的最近最少使用队列
int capacity = 0;
zlist_t *workers = zlist_new ();

while (true) {
    // 首先，路由任何来自工人的等待中的应答
    zmq_pollitem_t backends [] = {
        { localbe, 0, ZMQ_POLLIN, 0 },
        { cloudbe, 0, ZMQ_POLLIN, 0 }
    };
    // 如果我们没有工人，则无限期等待
    int rc = zmq_poll (backends, 2,
        capacity? 1000 * ZMQ_POLL_MSEC: -1);
    if (rc == -1)
        break;              // 中断

    // 处理来自本地工人的应答
    zmsg_t *msg = NULL;
    if (backends [0].revents & ZMQ_POLLIN) {
        msg = zmsg_recv (localbe);
        if (!msg)
            break;          // 中断
        zframe_t *identity = zmsg_unwrap (msg);
        zlist_append (workers, identity);
        capacity++;

        // 如果它是 READY，则不再进一步路由该消息
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
            zmsg_destroy (&msg);
    }
    // 否则就处理来自对等代理的应答
    else
    if (backends [1].revents & ZMQ_POLLIN) {
        msg = zmsg_recv (cloudbe);
        if (!msg)
            break;          // 中断
        // 我们不将对等代理身份用于任何用途
        zframe_t *identity = zmsg_unwrap (msg);
        zframe_destroy (&identity);
    }
    // 如果某个应答指向一个代理，则将它路由到云端
    for (argn = 2; msg && argn < argc; argn++) {
        char *data = (char *) zframe_data (zmsg_first (msg));
```

131

```
        size_t size = zframe_size (zmsg_first (msg));
        if (size == strlen (argv [argn])
        &&  memcmp (data, argv [argn], size) == 0)
            zmsg_send (&msg, cloudfe);
    }
    //  如果我们还需要将应答路由到客户端，就这么做
    if (msg)
        zmsg_send (&msg, localfe);
```

现在，只要我们还有工人处理能力，就路由尽可能多的客户端请求，如示例 3-26 所示。可以从我们的本地前端重新路由请求，但不能从云前端重新路由。现在，我们随机重新路由，只是为了测试的需要。在未来的版本中，我们将通过计算云端的处理能力来正确地处理这个问题。

示例3-26：本地和云端流原型（peering2.c）：客户端请求路由

132

```
    while (capacity) {
        zmq_pollitem_t frontends [] = {
            { localfe, 0, ZMQ_POLLIN, 0 },
            { cloudfe, 0, ZMQ_POLLIN, 0 }
        };
        rc = zmq_poll (frontends, 2, 0);
        assert (rc >= 0);
        int reroutable = 0;
        //  我们将先做对等代理，以防止饥饿
        if (frontends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudfe);
            reroutable = 0;
        }
        else
        if (frontends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localfe);
            reroutable = 1;
        }
        else
            break;      //  没有工作，则回到后端

        //  如果是可路由的，则在 20% 的时候发送到云端
        //  这里我们将通常使用云状态信息
        //
        if (reroutable && argc > 2 && randof (5) == 0) {
            //  路由到随机的代理节点
            int random_peer = randof (argc - 2) + 2;
            zmsg_pushmem (msg, argv [random_peer], strlen (argv [random_
              peer]));
            zmsg_send (&msg, cloudbe);
```

```
            }
            else {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                capacity--;
            }
        }
    }
    //  当我们完成工作时，就执行适当的清理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

例如，通过在两个窗口中启动代理的两个实例来执行此程序：

```
peering2 me you
peering2 you me
```

对这段代码的一些注释：

- 至少在 C 代码中，使用 zmsg 类会使工作变得更轻松，并使我们的代码更短。这显然是有效的抽象。如果你用 C 语言来构建 ØMQ 应用程序，那么应该使用 CZMQ。 133
- 因为没有得到来自节点的任何状态信息，所以我们天真地以为它们正在运行。当我们已经启动了所有代理时，代码提示我们确认。在实际情况下，我们不会向没有告诉它们确实存在的代理发送任何东西。

你能通过观察代码一直运行下去来证明它在正常工作。如果有任何错误路由的信息，客户端最终都会阻塞，而代理将停止输出跟踪信息。你可以通过清除任何一个代理来证明这一点。其他代理试图将请求发送到云中，而它的客户端会逐个地阻塞，等待一个应答。

总结

让我们把这些汇集成单个的封装。一如既往，我们将整个集群作为一个进程运行。我们将采用前两个例子，将它们合并成一个正常工作的设计，这可以让我们模拟任意数量的集群。

此代码的大小是将前两个原型的代码加在一起的大小，它包含 270 行代码。这对于一个包括客户端、工人和云工作负载分布的模拟集群是相当不错的。该代码会在下面的一系列示例中表示，从示例 3-27 开始。

示例3-27：完整的集群模拟（peering3.c）

```
//
//  代理对等节点模拟（第 3 部分）
//  状态和任务流的完整原型
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define WORKER_READY   "\001"      //  工人已准备就绪的信号

//  我们自己的名称，在实际工作中，这会在每个节点上配置
static char *self;
```

示例 3-28 显示了客户端的任务。它发出一个突发请求，然后休眠几秒钟。这模拟零星的活动，如果许多客户端都马上活跃，本地工人就会过载。客户端使用 REQ 套接字用于请求，并将统计信息推送到监控套接字上。

134 示例3-28：完整的集群模拟（peering3.c）：客户端任务

```
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "ipc://%s-monitor.ipc", self);

    while (true) {
        sleep (randof (5));
        int burst = randof (15);
        while (burst--) {
            char task_id [5];
            sprintf (task_id, "%04X", randof (0x10000));

            //  发送请求和随机的十六进制 ID
            zstr_send (client, task_id);

            //  对一个应答最多等候 10 秒，超过这个时间就抱怨
            zmq_pollitem_t pollset [1] = { { client, 0, ZMQ_POLLIN, 0 } };
```

```
            int rc = zmq_poll (pollset, 1, 10 * 1000 * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  中断

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
                    break;              //  中断
                //  工人要用我们的任务 ID 来回答我们
                assert (streq (reply, task_id));
                zstr_sendf (monitor, "%s", reply);
                free (reply);
            }
            else {
                zstr_sendf (monitor,
                    "E: CLIENT EXIT - lost task %s", task_id);
                return NULL;
            }
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

工人任务，它使用一个 REQ 套接字插入到负载均衡器，如示例 3-29 所示。那就是你已经在其他例子中看到过的相同的存根工人任务。

示例3-29：完整的集群模拟（peering3.c）：工人任务 135

```
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  告诉代理我们已准备好执行工作
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  处理到达的消息
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断
```

```
        //  工人会繁忙 0/1 秒
        sleep (randof (2));
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

主任务以设置它的所有套接字开始（参见示例 3-30）。本地前端与客户端交流，而本地后端与工人交流。云前端会与对等代理交流，就好像它们是客户端，而云后端也与对等代理交流，就好像它们是工人。状态后端定期发布状态信息，而状态前端订阅所有状态后端来收集这些信息。最后，我们使用一个 PULL 监控套接字来收集来自任务的可打印的消息。

示例3-30：完整的集群模拟（peering3.c）：主任务

```
int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是我们的对等节点名称
    //
    if (argc < 2) {
        printf ("syntax: peering3 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    zctx_t *ctx = zctx_new ();

    //  准备本地前端和后端
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);

136 void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  将云前端绑定到端点
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  将云后端连接到所有对等节点
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
```

```
zsockopt_set_identity (cloudbe, self);
int argn;
for (argn = 2; argn < argc; argn++) {
    char *peer = argv [argn];
    printf ("I: connecting to cloud frontend at '%s'\n", peer);
    zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
}
//  将状态后端绑定到端点
void *statebe = zsocket_new (ctx, ZMQ_PUB);
zsocket_bind (statebe, "ipc://%s-state.ipc", self);

//  将状态前端连接到所有对等节点
void *statefe = zsocket_new (ctx, ZMQ_SUB);
zsockopt_set_subscribe (statefe, "");
for (argn = 2; argn < argc; argn++) {
    char *peer = argv [argn];
    printf ("I: connecting to state backend at '%s'\n", peer);
    zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
}
//  准备监控套接字
void *monitor = zsocket_new (ctx, ZMQ_PULL);
zsocket_bind (monitor, "ipc://%s-monitor.ipc", self);
```

在绑定并连接所有的套接字后，我们启动子任务——工人和客户端，如示例 3-31 所示。

示例3-31：完整的集群模拟（peering3.c）：启动子任务

```
int worker_nbr;
for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
    zthread_new (worker_task, NULL);

//  启动本地客户端
int client_nbr;
for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
    zthread_new (client_task, NULL);

//  可用的工人的队列
int local_capacity = 0;
int cloud_capacity = 0;
zlist_t *workers = zlist_new ();
```

137

主循环（参见示例 3-32）有两个部分。首先，不管怎样，我们轮询工人和两个服务套接字（statefe 和 monitor）。如果没有准备就绪的工人，那么查看传入的请求就是没有意义的。这些请求可以保留在其内部 ØMQ 队列中。

示例3-32：完整的集群模拟（peering3.c）：主循环

```
while (true) {
    zmq_pollitem_t primary [] = {
        { localbe, 0, ZMQ_POLLIN, 0 },
        { cloudbe, 0, ZMQ_POLLIN, 0 },
        { statefe, 0, ZMQ_POLLIN, 0 },
        { monitor, 0, ZMQ_POLLIN, 0 }
    };
    //  如果没有准备就绪的工人，则无限期等待
    int rc = zmq_poll (primary, 4,
        local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
    if (rc == -1)
        break;              //  中断

    //  如果在这个迭代期间处理能力变化了，就追踪
    int previous = local_capacity;

    //  处理来自本地工人的应答
    zmsg_t *msg = NULL;

    if (primary [0].revents & ZMQ_POLLIN) {
        msg = zmsg_recv (localbe);
        if (!msg)
            break;          //  中断
        zframe_t *identity = zmsg_unwrap (msg);
        zlist_append (workers, identity);
        local_capacity++;

        //  如果该消息是READY，则不再进一步路由它
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
            zmsg_destroy (&msg);
    }
    //  或者处理来自对等代理的应答
    else
    if (primary [1].revents & ZMQ_POLLIN) {
        msg = zmsg_recv (cloudbe);
        if (!msg)
            break;          //  中断
        //  我们不把对等代理的身份用于任何用途
        zframe_t *identity = zmsg_unwrap (msg);
        zframe_destroy (&identity);
    }
    //  如果应答的地址指向一个代理，则把它路由到云中
    for (argn = 2; msg && argn < argc; argn++) {
```

138

```
        char *data = (char *) zframe_data (zmsg_first (msg));
        size_t size = zframe_size (zmsg_first (msg));
        if (size == strlen (argv [argn])
        &&  memcmp (data, argv [argn], size) == 0)
            zmsg_send (&msg, cloudfe);
    }
    //  如果我们仍然需要，则把应答路由到客户端
    if (msg)
        zmsg_send (&msg, localfe);
```

如果在 statefe 或 monitor 套接字上有输入消息，我们就可以马上处理这些消息，如示例 3-33 所示。

示例3-33：完整的集群模拟（peering3.c）：处理状态消息

```
    if (primary [2].revents & ZMQ_POLLIN) {
        char *peer = zstr_recv (statefe);
        char *status = zstr_recv (statefe);
        cloud_capacity = atoi (status);
        free (peer);
        free (status);
    }
    if (primary [3].revents & ZMQ_POLLIN) {
        char *status = zstr_recv (monitor);
        printf ("%s\n", status);
        free (status);
    }
```

现在，我们根据自己的处理能力来路由尽可能多的客户端请求，如示例 3-34 所示。如果有本地的处理能力，我们同时轮询 localfe 和 cloudfe。如果只有云处理能力，我们只是轮询 localfe。如果可能，我们会将任何请求都路由到本地，否则就将其路由到云中。

示例3-34：完整的集群模拟（peering3.c）：路由客户端请求

```
    while (local_capacity + cloud_capacity) {
        zmq_pollitem_t secondary [] = {
            { localfe, 0, ZMQ_POLLIN, 0 },
            { cloudfe, 0, ZMQ_POLLIN, 0 }
        };
        if (local_capacity)
            rc = zmq_poll (secondary, 2, 0);
        else
            rc = zmq_poll (secondary, 1, 0);
        assert (rc >= 0);

        if (secondary [0].revents & ZMQ_POLLIN)
            msg = zmsg_recv (localfe);
```

```
            else
            if (secondary [1].revents & ZMQ_POLLIN)
139
                msg = zmsg_recv (cloudfe);
            else
                break;      //  没有工作，回到主机

            if (local_capacity) {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                local_capacity--;
            }
            else {
                //  路由到随机的代理节点
                int random_peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [random_peer], strlen (argv [random_
                   peer]));
                zmsg_send (&msg, cloudbe);
            }
        }
```

我们将处理能力消息广播给其他对等节点，如示例 3-35 所示。为了减少多余的广播，只有当处理能力已经改变时我们才这样做。

示例3-35：完整的集群模拟（peering3.c）：广播处理能力

```
        if (local_capacity != previous) {
            //  我们把自己的身份附着在封包上
            zstr_sendm (statebe, self);
            //  广播新处理能力
            zstr_sendf (statebe, "%d", local_capacity);
        }
    }
    //  当完成后，执行适当的清理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

这是一个不简单的程序，我们花了大约一天时间才让它开始工作。下面这些都是重点：

- 客户端线程检测并报告一个失败的请求。它们通过轮询响应，如果一段时间（10 秒）后没有东西到达，就打印错误消息来完成这个工作。
- 客户端线程不直接输出消息，而是将消息发送到主循环收集（PULL）的监控套接字（PUSH），并由它输出。这是我们见过的用 ØMQ 套接字监控和记录日志的第一个用例，这个用例是很大的，稍后还会来讨论它。
- 客户端模拟负载变化来使得集群负载在随机的时刻达到 100%，以便让任务转移到云中。客户端和工人的数量与在客户端和工作线程的延迟对此进行控制。请随时用 140
它们来演练，看看你是否能做出更逼真的模拟。
- 主循环使用两个轮询集。它其实可以使用三个：信息、后端和前端。正如在前面的原型中说明的，如果没有后端的处理能力，获取一个前端的消息是没有意义的。

下面这些都是这个程序在开发过程中出现的问题：

- 客户端会冻结，由于请求或应答在某个地方丢失。回想一下，ROUTER 套接字会丢弃无法路由的消息。这里的第一个策略是修改客户端线程来检测和报告这些问题。其次，我在主循环的每一个接收后面和每一个发送前面都执行 `zmsg_dump()` 调用，直到弄清问题的根源。
- 主循环错误地从多个准备就绪的套接字读取，这导致丢失第一条消息。我通过只从第一个准备就绪的套接字读取修复了这个问题。
- `zmsg` 类未将 UUID 作为 C 字符串正确编码，这导致包含 0 字节的 UUID 被损坏。我通过将 `zmsg` 修改为把 UUID 编码为可打印的十六进制字符串修复了这个问题。

这个模拟器无法侦测到云对等节点的消失。如果你启动几个对等节点并停止其中的一个，而它正在把处理能力广播给其他节点，那么它们会不停地发送工作给它，即使在它消失以后。你可以试试这个，会发现抱怨丢失了请求的客户端。该解决方案是双重的。第一，只将处理能力信息保留很短的时间，所以，如果某个对等节点消失，其处理能力就会被快速设置为零。第二，在请求 - 应答链中加入可靠性。我们将在下一章着重处理可靠性。

第 4 章 141

可靠的请求-应答模式

第 3 章我们采用工作的例子介绍了 ØMQ 的请求 - 应答模式的高级使用。本章着眼于可靠性的一般问题，并在 ØMQ 的核心请求 - 应答模式的基础上建立一套可靠的消息传递模式。

在本章中，我们在很大程度上专注于用户空间的请求 - 应答模式，它们是可重用的模型，可以帮助你设计自己的 ØMQ 架构。

- 懒惰海盗模式：来自客户端的可靠的请求 - 应答。
- 简单海盗模式：使用负载均衡的可靠的请求 - 应答。
- 偏执海盗模式：使用信号检测的可靠的请求 - 应答。
- 管家模式：面向服务的可靠排队。
- 泰坦尼克模式：基于磁盘 / 断开连接的可靠排队。
- 双星模式：主备份服务器故障转移。
- 自由职业者模式：缺少代理的可靠的请求 - 应答。

什么是“可靠性”

大多数人谈到“可靠性”时并不知道它的确切含义。我们只能从故障的角度来定义可靠性。也就是说，如果我们可以处理一组特定的被明确定义和理解的故障，那么我们对于这些故障是可靠的。不会多，也不会少。所以，让我们来看看在分布式 ØMQ 应用程序中的故障的可能原因，粗略地按概率大小降序排列：

1. 应用程序代码是最坏的罪犯。它可能会崩溃并退出，冻结并停止响应输入，运行速度对于其输入而言太慢，耗尽所有的内存等。
2. 系统代码（如我们使用 ØMQ 编写的代理）可能出于和应用程序代码同样的原因失败。 142

系统代码应该比应用程序代码更可靠，但它仍然可能崩溃和毁坏，尤其是，如果它试图对慢速客户端的消息排队，就会用尽内存。

3. 消息队列可能会溢出，这通常会发生在一个已经学会了野蛮地处理慢速客户端的系统代码中。当队列溢出时，它开始丢弃消息，所以我们就会“丢失”消息。
4. 网络可能会失效（例如，无线网络被关闭或超出范围）。在这种情况下，ØMQ 会自动重新连接，但在此期间，消息可能会丢失。
5. 硬件可能会失效，并造成在这台机器上运行的所有进程的失败。
6. 网络可能会因外来的途径失败，例如，一个交换机的某些端口可能会坏掉，而网络的一部分会变得无法访问。
7. 整个数据中心可能毁于雷电、地震、火灾，或更常见的电力或冷却故障。

针对所有这些可能的故障，制作一个完全可靠的软件系统是一项非常困难和昂贵的工作，这超出了本书的范围。

因为上述列表中的前 5 种情况覆盖了除大公司以外 99.9%的可能性最大的实际需求（根据我刚刚做的一项高度科学的研究，78%的统计数据都由这些情况组成），那就是我们将在这里研究的内容。

可靠性设计

因此，为了使事情简单化，可靠性就是“在代码冻结或崩溃时，让事情保持正常工作”，我们将缩短“代码崩溃的时间”的一种情况。不过，我们要保持正常的不仅仅是消息，还有更复杂的东西。我们需要研究每种核心 ØMQ 消息传递的模式，看看如何使它即使在代码崩溃时，也能工作（如果我们可以做到）。

让我们逐一遍历这些模式。

请求 - 应答

如果正在处理请求的服务器死机，客户端可以发现这一点，因为它不会得到回答了。然后，它可以采取在一怒之下放弃、等待、稍后再试以及发现另一台服务器等处理方式；至于客户端死机，现在我们可以把它当作“别人的问题”而不去管它。

143 *发布 - 订阅*

如果客户端死机（它已经得到了一些数据），服务器将不会知道这一点。因为发布 - 订阅不从客户端发回任何信息给服务器。但是，客户端可以通过其他的频带与服务器联系，例如，通过请求 - 应答说，“请重新发送我错过了的全部消息。”至于服务器死机，不在这里的讨论范围之内。订阅者也可以自行验证它们是否运行过慢，并且如果它们确实运行过慢，就可以采取行动（例如，警告操作员并崩溃）。

流水线

如果工人崩溃（在工作时），发生器不会知道这件事。因为流水线，如同发布 - 订阅和时间的齿轮，只能在一个方向上运行。但下游收集器可以检测到一个任务没有完成，并发送一个消息给发生器说："嘿，重新发送任务 324！" 如果发生器或收集器坏掉，那么任何最初发送批处理工作的上游客户端都会厌倦等待，而重新发送整个任务。这种方式虽不优雅，但系统代码应该真的不应因这个原因而频繁死掉。

在本章中，我们将只把注意力集中在请求 - 应答上，这是最容易实现的可靠的消息传递。

对于处理最常见类型的故障，基本的请求 - 应答模式（一个 REQ 客户端套接字对一个 REP 服务器套接字执行阻塞的发送 / 接收）的得分很低。如果在处理请求时服务器崩溃，客户端就会挂起，直到永远。同样，如果网络丢失请求或应答，客户端也会挂起，直到永远。

请求 - 应答仍然比 TCP 好很多，这要归功于 ØMQ 具有默默地重新连接节点、负载均衡消息等能力。但它对于实际的工作仍然不够好。你可以真正信任基本请求 - 应答模式的唯一情况是：在同一个进程的两个线程之间，那里没有网络或独立的服务器进程可以失效。

然而，随着添加一些额外的工作，这个不起眼的模式为跨越分布式网络的实际工作奠定了良好基础，而我们得到了一组可靠的请求 - 应答（RRR）的模式，我喜欢将它们称为海盗模式（我希望你最终会得到乐趣）。

根据我的经验，将客户端连接到服务器大致有三种方式。每种方式都需要一个具体做法来保持可靠性：

1. 多个客户端与一台服务器直接交流。用例：客户端需要与一台众所周知的服务器交流。我们要处理的目标故障类型：服务器崩溃并重新启动，以及网络断开连接。
2. 多个客户端与一台将工作分配给多个工人的代理服务器交流。用例：面向服务的事务处理。我们要处理的目标故障类型：工人崩溃和重启、工人忙于循环、工人超负荷、队列崩溃并重新启动，以及网络断开连接。
3. 多个客户端与多台服务器交流，而没有中间层代理。用例：分布式服务，如名称解析。我们要处理的目标故障类型：服务崩溃并重新启动、服务忙于循环、服务过载，以及网络断开连接。 144

每一种方法都有它的取舍，往往你会混合使用它们。我们来详细研究这三种方法。

客户端可靠性（懒惰海盗模式）

对客户端稍做改变，我们就可以得到很简单的、可靠的请求 - 应答，我们称之为懒惰海

盗模式（参见图 4-1）。我们不是执行一个阻塞的接收，而是：

- 仅当确信应答已经到达时，才轮询 REQ 套接字并接收它的应答。
- 如果在超时时间内没有应答到达，则重新发送一个请求。
- 如果多次请求后还是没有应答，则放弃事务。

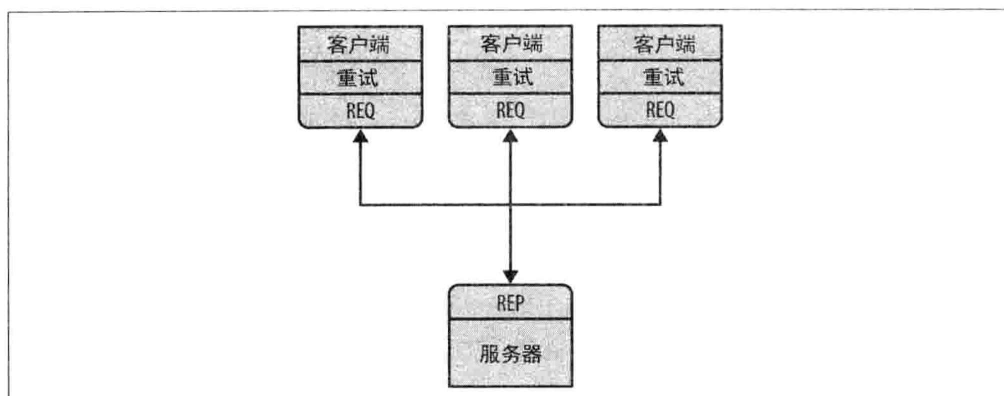


图4-1：懒惰海盗模式

如果在除严格的发送/接收方式以外的任何其他方式中尝试使用 REQ 套接字，我们就会得到一个错误（在技术上，该 REQ 套接字实现了一个小的有限状态机来执行发送/接收乒乓信号，所以错误代码被称为“EFSM（扩展有限状态机）”）。当我们要在海盗模式中使用 REQ 时，这有点烦人，因为在得到应答之前，可能会发送几个请求，正如你可以在示例 4-1 中看到的。相当不错的蛮力解决方案是在发生一个错误后，关闭并重新打开 REQ 套接字。

示例4-1：懒惰海盗客户端（lpclient.c）

```
//
//  懒惰海盗客户端
//  使用 zmq_poll 执行安全的请求 - 应答
145 //  要运行它，启动 lpserver，然后再随机清除/重启它
//
#include "czmq.h"

#define REQUEST_TIMEOUT     2500    //  毫秒（> 1000！）
#define REQUEST_RETRIES     3       //  放弃前重试次数
#define SERVER_ENDPOINT     "tcp://localhost:5555"

int main (void)
{
```

```
    zctx_t *ctx = zctx_new ();
    printf ("I: connecting to server...\n");
    void *client = zsocket_new (ctx, ZMQ_REQ);
    assert (client);
    zsocket_connect (client, SERVER_ENDPOINT);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    while (retries_left && !zctx_interrupted) {
        //  我们发送一个请求，然后努力去获取一个应答
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            //  以超时时间轮询套接字以得到一个应答
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  中断
```

示例 4-2 显示了如何处理服务器响应，并且如果应答是有效的，则退出循环。如果没有收到应答，关闭客户端套接字，并重新发送请求。我们尝试很多次后终于放弃了。

示例4-2：懒惰海盗客户端（lpclient.c）：服务器进程应答

```
            if (items [0].revents & ZMQ_POLLIN) {
                //  我们从服务器获得一个应答，它必须匹配请求的序号
                char *reply = zstr_recv (client);
                if (!reply)
                    break;      //  中断
                if (atoi (reply) == sequence) {
                    printf ("I: server replied OK (%s)\n", reply);
                    retries_left = REQUEST_RETRIES;
                    expect_reply = 0;
                }
                else
                    printf ("E: malformed reply from server: %s\n",
                        reply);

                free (reply);
            }
            else
            if (--retries_left == 0) {
                printf ("E: server seems to be offline, abandoning\n");
```

146

```
                break;
            }
            else {
                printf ("W: no response from server, retrying...\n");
                //  旧套接字是令人迷惑的，关闭它并打开一个新套接字
                zsocket_destroy (ctx, client);
                printf ("I: reconnecting to server...\n");
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, SERVER_ENDPOINT);
                //  在新套接字上再次发送请求
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

我们连同匹配的服务器一起运行这个程序，如示例 4-3 所示。

示例4-3：懒惰海盗服务器（lpserver.c）

```
//
//  懒惰海盗服务器
//  绑定 REQ 套接字到 tcp://*:5555
//  类似于 hwserver，除了：
//   - 按原样回应请求
//   - 随机地慢速运行，或退出以模拟服务器崩溃。
//
#include "zhelpers.h"

int main (void)
{
    srandom ((unsigned) time (NULL));

    void *context = zmq_ctx_new ();
    void *server = zmq_socket (context, ZMQ_REP);
    zmq_bind (server, "tcp://*:5555");

    int cycles = 0;
    while (1) {
        char *request = s_recv (server);
        cycles++;

        //  模拟各种问题，在几个周期后
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: simulating a crash\n");
```

```
            break;
        }
        else
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: simulating CPU overload\n");
            sleep (2);
        }
        printf ("I: normal request (%s)\n", request);
        sleep (1);              //  做某些繁重的工作
        s_send (server, request);
        free (request);
    }
    zmq_close (server);
    zmq_ctx_destroy (context);
    return 0;
}
```


为了运行这个测试用例，需要分别在两个控制台窗口中启动客户端和服务器。经过几个消息后，服务器会随机地胡作非为。你可以检查客户端的响应。下面是服务器的典型输出：

```
I: normal request (1)
I: normal request (2)
I: normal request (3)
I: simulating CPU overload
I: normal request (4)
I: simulating a crash
```

而下面是客户端的响应：

```
I: connecting to server...
I: server replied OK (1)
I: server replied OK (2)
I: server replied OK (3)
W: no response from server, retrying...
I: connecting to server...
W: no response from server, retrying...
I: connecting to server...
E: server seems to be offline, abandoning
```

客户端按顺序排列每个消息，并检查应答是否完全按顺序返回：没有任何请求或应答丢失，也没有应答多次返回或混乱地返回。运行测试几次，直到你确信这种机制实际上能够正常工作。在生产应用程序中不需要序列号，它们只是帮我们相信我们的设计是正确的。

客户端使用 REQ 套接字，它执行蛮力关闭 / 重新开启，因为 REQ 套接字施加了严格的

发送 / 接收周期。你可能想使用一个 DEALER 代替它，但这不会是一个很好的决定。首先，这将意味着模仿 REQ 对封包做处理的秘密武器（如果忘记了那是什么，那么它是一个你不必做这件事的好兆头）。其次，这将意味着你可能收到意料之外的应答。

148 当我们有一组客户端与单台服务器交流时，只有在客户端工作时才能处理失败。这样的设计可以处理服务器崩溃，但只有当恢复意味着重新启动同一台服务器时才可以这么做。如果有一个永久的错误，比如服务器硬件的电源损坏，这种方法是行不通的。因为在服务器上的应用程序代码通常是任何架构失败的最大来源，依赖于一台服务器不是一个好主意。

因此，这么做的优缺点如下所述。

- 优点：易于理解和实施。
- 优点：能与现有的客户端和服务器应用程序代码轻松地配合使用。
- 优点：ØMQ 自动重试实际的重新连接，直到它工作为止。
- 缺点：不能执行到备份或备用服务器的故障切换。

基本可靠队列（简单海盗模式）

我们的第二种做法是，带有队列代理的懒惰海盗模式，这个代理让我们可以透明地与多台服务器交流，我们可以更准确地称这些服务器为“工人”，我们会分阶段开发这个模式，从最简化的工作模式，即简单海盗模式开始。

在所有这些海盗模式中，工人都是无状态的。如果应用程序需要一些共享的状态，比如一个共享的数据库，我们在设计消息框架时不知道这件事。有一个队列代理意味着工人可以来去自如，客户端不知道关于它的任何事。如果一个工人崩溃，另一个工人就接管它。这是一个不错的且简单的拓扑结构，它只有一个真正的弱点：中央队列本身，它可能成为一个要管理的问题，并且是一个单点故障。

队列代理的基础是来自第 3 章的负载均衡代理。要处理崩溃或阻塞的工人，我们需要做的*最起码*的工作是什么呢？事实证明，这个工作少得令人惊讶。我们在客户端已经有一个重试机制，因此使用负载均衡模式将工作得很好。这符合 ØMQ 的理念，我们可以通过在中间层插入代理人来扩展类似于请求 - 应答的对等网络模式（参见图 4-2）。

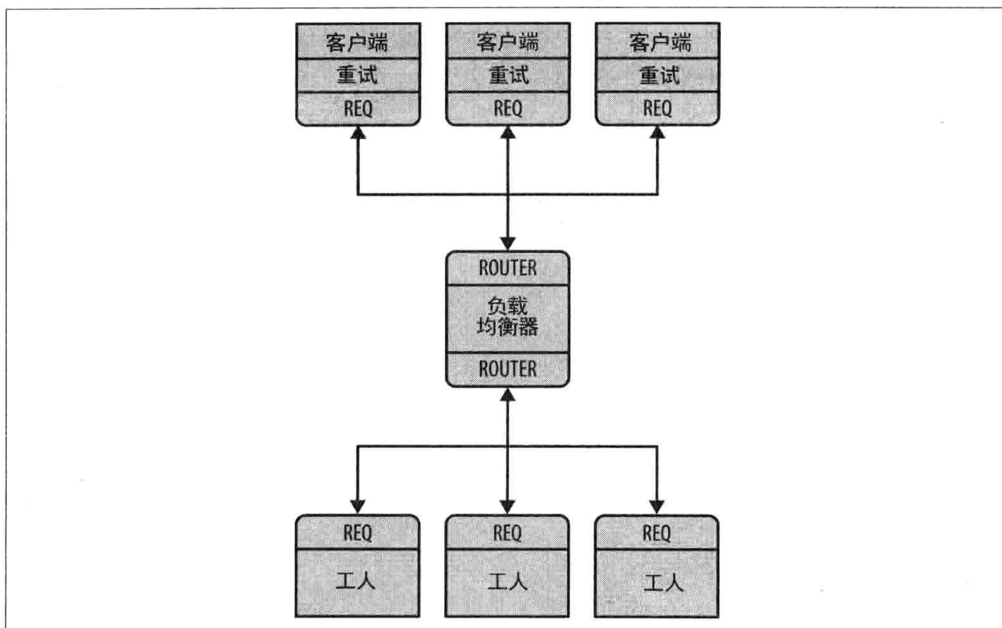


图4-2：简单海盗模式

我们不需要特殊的客户端，仍然使用懒惰海盗客户端。示例 4-4 给出的是队列，这与负载均衡代理的主要任务是相同的。 149

示例4-4：简单海盗队列（spqueue.c）

```
//
//  简单海盗代理
//  这与负载均衡模式是相同的，不带可靠性
//  机制。它依赖于客户端来恢复。始终运行。
//
#include "czmq.h"

#define WORKER_READY   "\001"      //  工人准备就绪的信号

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  用于客户端
    zsocket_bind (backend,  "tcp://*:5556");    //  用于工人

    //  可用的工人的队列
```

```
    zlist_t *workers = zlist_new ();

    // 本示例的主体与 lbbroker2 完全相同
```
150
```
    ...
}
```

示例 4-5 显示了工人的代码，这取自懒惰海盗服务器并改写它来适应负载均衡模式（使用 REQ“准备就绪”的信号）。

示例4-5：简单海盗工人（spworker.c）

```
//
// 简单海盗工人
// 连接 REQ 套接字到 tcp://*:5556
// 实现了负载均衡的工人部分
//
#include "czmq.h"
#define WORKER_READY   "\001"      // 工人准备就绪的信号

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);

    // 设置随机的身份以便于跟踪
    srandom ((unsigned) time (NULL));
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen (identity));
    zsocket_connect (worker, "tcp://localhost:5556");

    // 告诉代理我们已准备好接受工作
    printf ("I: (%s) worker ready\n", identity);
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    int cycles = 0;
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              // 中断

        // 模拟各种问题，在几个周期后
        cycles++;
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating a crash\n", identity);
```

```
            zmsg_destroy (&msg);
            break;
        }
        else
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating CPU overload\n", identity);
            sleep (3);
            if (zctx_interrupted)
                break;
        }                                                        151
        printf ("I: (%s) normal reply\n", identity);
        sleep (1);              //  做某些繁重的工作
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return 0;
}
```

为了验证这一点，请以任何顺序启动少量工人、一个懒惰海盗客户端，以及队列。你会看到，工人们最终全部崩溃，并在客户端重试，然后放弃。队列永远不会停止，你可能会不胜其烦地重新启动工人及客户端。这种模式适用于任何数量的客户端和工人。

健壮的可靠队列（偏执海盗模式）

简单海盗队列模式工作得很好，特别是因为它只是现有的两个模式的一个组合。不过，它也有一些缺点：

- 它在面临队列崩溃和重启时不够健壮。客户端将恢复，但工人不会。虽然 ØMQ 会自动重新连接工人的套接字，但直到新启动的队列已连接前，工人都不会发出准备就绪的信号，因此它们不存在。为了解决这个问题，需要在队列与工人之间做信号检测，让工人可以检测到队列已经死机。
- 队列不检测工人的故障，因此，如果一个工人在空闲时崩溃，除非队列给它发送一个请求，否则队列不能从工作队列中将其删除。客户端无谓地等待和重试。虽然这不是一个严重的问题，但不太好。为了使这项工作正常，我们需要在工人与队列之间做信号检测，让队列可以在任何阶段都检测到某个丢失的工人。

我们会在一个适当迂腐的偏执海盗模式中解决这些问题。

我们以前对工人使用一个 REQ 套接字。对于偏执海盗工人，我们将切换到一个 DEALER 套接字（参见图 4-3）。这让我们具有在任何时候发送和接收消息的优势，而不是 REQ 强加的步调一致地发送 / 接收。DEALER 的缺点是，必须自己做我们的封包管

理（请重读第 3 章复习这个概念的背景）。

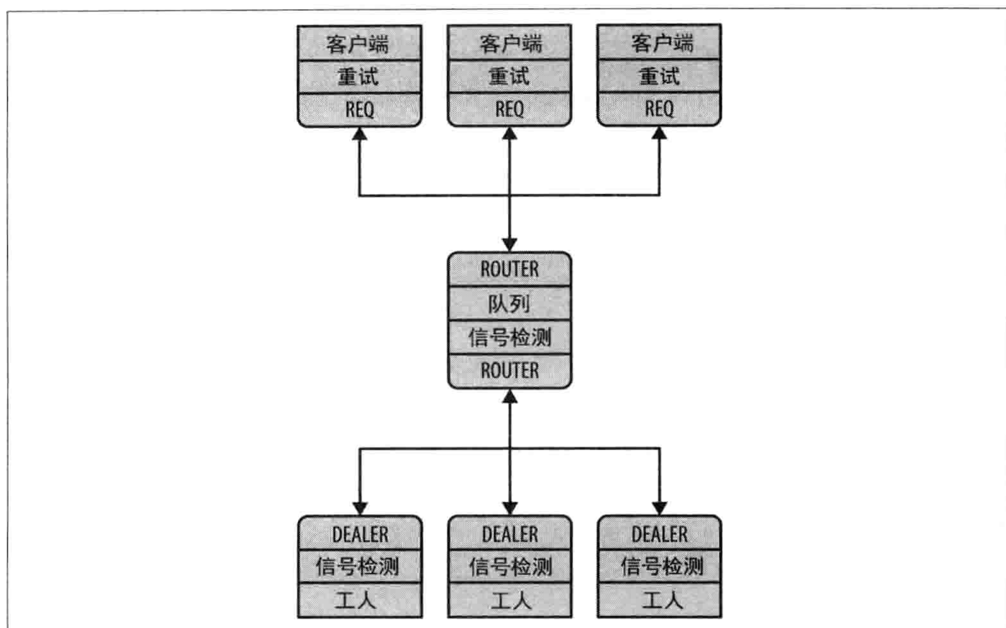


图4-3：偏执海盗模式

152 我们仍然使用懒惰海盗客户端。偏执海盗队列代理如示例 4-6 所示。

示例4-6：偏执海盗队列（ppqueue.c）

```
//
//  偏执海盗队列
//
#include "czmq.h"

#define HEARTBEAT_LIVENESS  3       //  3-5 是合理的
#define HEARTBEAT_INTERVAL  1000    //  毫秒

//  偏执海盗协议常量
#define PPP_READY       "\001"      //  工人准备就绪的信号
#define PPP_HEARTBEAT   "\002"      //  工人信号检测的信号
```

示例 4-7 定义了工人类：一个结构和一组函数，它们作为构造函数、析构函数，以及对工人对象的方法。

示例4-7：偏执海盗队列（ppqueue.c）：工人类结构

```
typedef struct {
```

```
    zframe_t *identity;          //  工人的身份
    char *id_string;             //  可打印的身份
    int64_t expiry;              //  在这个时间过期
} worker_t;
```

153

```
//  构造新工人
static worker_t *
s_worker_new (zframe_t *identity)
{
    worker_t *self = (worker_t *) zmalloc (sizeof (worker_t));
    self->identity = identity;
    self->id_string = zframe_strdup (identity);
    self->expiry = zclock_time () + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS;
    return self;
}

//  销毁指定的工人对象，包括身份帧
static void
s_worker_destroy (worker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        worker_t *self = *self_p;
        zframe_destroy (&self->identity);
        free (self->id_string);
        free (self);
        *self_p = NULL;
    }
}
```

ready 方法（参见示例 4-8）在准备就绪列表的末尾放置一个工人。

示例4-8：偏执海盗队列（ppqueue.c）：工人准备方法

```
static void
s_worker_ready (worker_t *self, zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (streq (self->id_string, worker->id_string)) {
            zlist_remove (workers, worker);
            s_worker_destroy (&worker);
            break;
        }
        worker = (worker_t *) zlist_next (workers);
    }
```

```
    zlist_append (workers, self);
}
```

next 方法如示例 4-9 所示，返回下一个可用工人的身份。

示例4-9：偏执海盗队列（ppqueue.c）：获取下一个可用工人的方法

154

```
static zframe_t *
s_workers_next (zlist_t *workers)
{
    worker_t *worker = zlist_pop (workers);
    assert (worker);
    zframe_t *frame = worker->identity;
    worker->identity = NULL;
    s_worker_destroy (&worker);
    return frame;
}
```

purge 方法（参见示例 4-10），寻找并清除过期的工人。我们保存从最旧到最新的工人，所以我们停在第一个“活着”的工人上。

示例4-10：偏执海盗队列（ppqueue.c）：清除过期工人的方法

```
static void
s_workers_purge (zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              //  工人是“活着”的，我们在这里就完成了

        zlist_remove (workers, worker);
        s_worker_destroy (&worker);
        worker = (worker_t *) zlist_first (workers);
    }
}
```

主任务是带有对工人做信号检测的负载均衡器，所以我们可以检测出崩溃或阻塞的工作任务，如示例 4-11 所示。

示例4-11：偏执海盗队列（ppqueue.c）：主任务

```
int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  用于客户端
    zsocket_bind (backend,  "tcp://*:5556");    //  用于工人
```

```
//  可用的工人的列表
zlist_t *workers = zlist_new ();

//  以常规时间间隔发出检测信号
uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

while (true) {
    zmq_pollitem_t items [] = {
        { backend,  0, ZMQ_POLLIN, 0 },
        { frontend, 0, ZMQ_POLLIN, 0 }
    };
    //  只在我们有可用的工人时轮询前台
    int rc = zmq_poll (items, zlist_size (workers)? 2: 1,
        HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
    if (rc == -1)
        break;              //  中断

    //  在后台处理工人活动
    if (items [0].revents & ZMQ_POLLIN) {
        //  使用工人身份来做负载均衡
        zmsg_t *msg = zmsg_recv (backend);
        if (!msg)
            break;          //  中断

        //  来自工人的任何存活信号表示它已准备就绪
        zframe_t *identity = zmsg_unwrap (msg);
        worker_t *worker = s_worker_new (identity);
        s_worker_ready (worker, workers);

        //  验证控制消息，或给客户端返回应答
        if (zmsg_size (msg) == 1) {
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), PPP_READY, 1)
            && memcmp (zframe_data (frame), PPP_HEARTBEAT, 1)) {
                printf ("E: invalid message from worker");
                zmsg_dump (msg);
            }
            zmsg_destroy (&msg);
        }
        else
            zmsg_send (&msg, frontend);
    }
    if (items [1].revents & ZMQ_POLLIN) {
        //  现在获取下一个客户端请求，路由到下一个工人
```

155

```
            zmsg_t *msg = zmsg_recv (frontend);
            if (!msg)
                break;          //  中断
            zmsg_push (msg, s_workers_next (workers));
            zmsg_send (&msg, backend);
        }
```

我们在任何套接字活动后，处理信号检测。如示例 4-12 所示，首先，如果时间到了，我们发送检测信号给任何空闲的工人，随后清除任何崩溃的工人。

示例4-12：偏执海盗队列（ppqueue.c）：处理信号检测

```
        if (zclock_time () >= heartbeat_at) {
            worker_t *worker = (worker_t *) zlist_first (workers);
            while (worker) {
                zframe_send (&worker->identity, backend,
                             ZFRAME_REUSE + ZFRAME_MORE);
                zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
                zframe_send (&frame, backend, 0);
                worker = (worker_t *) zlist_next (workers);
            }
156         heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
        s_workers_purge (workers);
    }

    //  当我们完成后，正确地执行清理
    while (zlist_size (workers)) {
        worker_t *worker = (worker_t *) zlist_pop (workers);
        s_worker_destroy (&worker);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

队列使用检测工人的信号扩展了负载均衡模式。信号检测是那种看起来“简单”，但可能很难正确地实现的事情。我们会在接下来的章节更多地解释这一点，现在，回到代码上来。

让我们来看看示例 4-13 中的偏执海盗工人。

示例4-13：偏执海盗工人（ppworker.c）

```
//
//  偏执海盗工人
//
#include "czmq.h"
```

```
#define HEARTBEAT_LIVENESS  3        //  3-5 是合理的
#define HEARTBEAT_INTERVAL  1000     //  毫秒
#define INTERVAL_INIT       1000     //  重新连接的初始值
#define INTERVAL_MAX       32000     //  在指数级退避后

//  偏执海盗协议常量
#define PPP_READY       "\001"      //  工人准备就绪的信号
#define PPP_HEARTBEAT   "\002"      //  工人信号检测的信号

//  返回一个新配置的套接字的辅助函数
//  连接到偏执海盗队列

static void *
s_worker_socket (zctx_t *ctx) {
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");

    //  告诉队列我们已经准备好执行工作
    printf ("I: worker ready\n");
    zframe_t *frame = zframe_new (PPP_READY, 1);
    zframe_send (&frame, worker, 0);

    return worker;
}
```

我们拥有一个任务，它实现了偏执海盗协议（PPP）的工人一方。示例4-14中的信号检测代码可以让工人检测该队列是否已经崩溃了，反之亦然。 157

示例4-14：偏执海盗工人（ppworker.c）：主任务

```
int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = s_worker_socket (ctx);

    //  如果活跃度触及零，就认为队列已经断开连接
    size_t liveness = HEARTBEAT_LIVENESS;
    size_t interval = INTERVAL_INIT;

    //  以常规时间间隔发出检测信号
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    srandom ((unsigned) time (NULL));
    int cycles = 0;
    while (true) {
```

```
        zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        if (items [0].revents & ZMQ_POLLIN) {
            //  获取消息
            //  -3 部分封包 + 内容 -> 请求
            //  -1 部分 HEARTBEAT-> 信号检测
            zmsg_t *msg = zmsg_recv (worker);
            if (!msg)
                break;          //  中断
```

为了测试这个队列实现的健壮性，我们模拟各种典型问题，如工人崩溃或运行速度非常缓慢。为此，我们在经过几个周期后做这件事，以示这样的架构可以首先启动和运行起来。问题模拟代码在示例 4-15 中。

示例4-15：偏执海盗工人（ppworker.c）：模拟问题

```
                cycles++;
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: simulating a crash\n");
                    zmsg_destroy (&msg);
                    break;
                }
                else
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: simulating CPU overload\n");
                    sleep (3);
                    if (zctx_interrupted)
158                     break;
                }
                printf ("I: normal reply\n");
                zmsg_send (&msg, worker);
                liveness = HEARTBEAT_LIVENESS;
                sleep (1);              //  做某些繁重的工作
                if (zctx_interrupted)
                    break;
            }
            else
```

当我们从队列中获得一个信号检测消息时，这意味着队列（或更确切地说，是最近）还活着，所以我们必须重新设置我们的活跃度指标。示例 4-16 中的代码负责处理信号检测。

示例4-16：偏执海盗工人（ppworker.c）：处理信号检测

```
                zframe_t *frame = zmsg_first (msg);
```

```
                if (memcmp (zframe_data (frame), PPP_HEARTBEAT, 1) == 0)
                    liveness = HEARTBEAT_LIVENESS;
                else {
                    printf ("E: invalid message\n");
                    zmsg_dump (msg);
                }
                zmsg_destroy (&msg);
            }
            else {
                printf ("E: invalid message\n");
                zmsg_dump (msg);
            }
            interval = INTERVAL_INIT;
        }
        else
```

如果过一段时间队列还没有发给我们检测信号，我们就销毁套接字，然后重新连接，如示例 4-17 所示。这是丢弃可能在此期间发送的任何消息的最简单和最野蛮的方式。

示例4-17：偏执海盗工人（ppworker.c）：检测一个崩溃队列

```
            printf ("W: heartbeat failure, can't reach queue\n");
            printf ("W: reconnecting in %zd msec...\n", interval);
            zclock_sleep (interval);

            if (interval < INTERVAL_MAX)
                interval *= 2;
            zsocket_destroy (ctx, worker);
            worker = s_worker_socket (ctx);
            liveness = HEARTBEAT_LIVENESS;
        }

        //  如果到时间了就向队列发送检测信号
        if (zclock_time () > heartbeat_at) {
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
            printf ("I: worker heartbeat\n");
            zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
            zframe_send (&frame, worker, 0);
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

159

关于这个例子的一些注释：

- 和以前一样，该代码包括对故障的模拟。这使得它（a）非常难以调试，并且（b）

对于重用是危险的。当你想调试这段代码时，可以禁用故障模拟。

- 工人使用类似于我们为懒惰海盗客户端设计的重新连接策略，但有两个主要区别：它执行指数级退避，而且无限期重试（而客户端会在报告失败之前重试几次）。

可以通过使用一个如下所示的脚本在客户端、队列和工人上尝试：

```
ppqueue &
for i in 1 2 3 4; do
    ppworker &
    sleep 1
done
lpclient &
```

当工人们模拟崩溃时，你应该看到它们一个个死掉，并且客户端最终会放弃。你可以停止并重新启动队列，客户端和工人将重新连接和执行。不管你对队列和工人做了什么，客户端将永远不会得到无序的应答：要么整个链条都能工作，要么客户端放弃。

信号检测

信号检测用来解决确定节点是活着还是死掉的问题。这不是特定于 ØMQ 的问题。TCP 有一个漫长的超时时间（30 分钟左右），这意味着它可能无法知道节点是否已经死掉、被断开，或带着一箱伏特加、一个红发女郎和一大笔钱走在周末前往布拉格的路上。

正确地执行信号检测是不容易的。在编写偏执海盗的例子时，我花了大约五个小时才使信号检测能够正常工作。请求 - 应答链的其余部分大约花了 10 分钟。“假故障”是特别容易发生的，也就是说，是因为信号检测不正常发送，所以节点才会认为它们已断开连接。

160 在本节中，我们将看看人们使用 ØMQ 来检测信号的三个主要解决方案。

置若罔闻地将它关闭

最常见的方法就是不做信号检测并抱乐观态度。即使不是大多数，也有许多 ØMQ 应用程序是这么做的。ØMQ 通过在许多情况下隐藏节点来鼓励这一做法。这种做法会导致什么问题呢？

- 当我们在跟踪节点的应用程序中使用 ROUTER 套接字，当节点断开并重新连接时，应用程序会产生内存泄漏（应用程序用来保存每个节点的资源），并变得越来越慢。
- 当使用以 SUB 或 DEALER 为基础的数据接收者，我们不能分辨出好沉默（没有数据）和坏沉默（另一端已崩溃）之间的差异。例如，当收件人知道对方已经崩溃时，它可以切换到备份路由。

- 在某些网络中，如果我们使用长时间保持沉默的一个 TCP 连接，它只会是崩溃。发送某些东西（从技术上来说，“保持活动”不止是信号检测）将使网络保持连通。

单向信号检测

第二个选择是，大约每隔一秒从每个节点发送信号检测消息给对等节点。当一个节点在某个超时时间（通常情况下是几秒钟）内没有从另一个节点听到任何东西，它将把该对等节点当作已崩溃。听起来不错，对吧？可悲的是，不行。这在某些情况下能工作，但在其他时候有讨厌的边缘情况。

对于发布 - 订阅模式，这种方法是行得通的，它是你可以使用的唯一模式。SUB 套接字不能反过来与 PUB 套接字交流，但 PUB 套接字可以愉快地发送“我还活着”的消息给它们的订阅者。

作为一种优化，可以只在没有实际数据发送时发送信号检测。此外，还可以逐步以较长的时间间隔发送信号检测，如果网络活动是一个问题（例如，在移动网络中，活动会消耗电池电量）。只要接收方可以检测故障（马上停止活动），这很好。

下面是这种设计的典型问题：

- 当我们发送大量数据时，因为信号检测将被推迟到数据的背后，所以它可能是不准确的。如果信号检测延迟，你可能会由于网络拥塞而得到虚假的超时和断开。因此，应当始终把任何输入数据都作为信号检测，而不论发送者是否优化了信号检测。
- 虽然发布 - 订阅模式将丢弃消失的接收者的消息，但 PUSH 和 DEALER 套接字会对 161
  它们排队。所以，如果你已经给死掉的节点发送了信号检测，而它起死回生了，那么它会得到你发送的所有检测信号，这可能是数千个消息。哇，哇！
- 这种设计假定整个网络的信号检测超时是相同的。但是，这是不准确的。有些节点想要非常积极的信号检测，以便迅速发现故障。有的想要很轻松的信号检测，比如说为了让沉睡的网络待机，并节省电力。

乒乓信号检测

第三个选择是使用一个乒乓对话框。一个节点发送一个 *ping* 命令到另一个节点，后者以一个 *pong* 命令回复。无论是哪个命令都没有任何的有效载荷。*ping* 和 *pong* 是不相关的。因为在某些网络中，“客户端”和“服务器”的角色是任意的，我们通常会指定，任意一方实际上都可以发送一个 *ping* 并期望得到 *pong* 响应。然而，由于对于动态客户端，超时依赖于已知的最好的网络拓扑结构，所以通常是客户端在 *ping* 服务器。

这适用于所有基于 ROUTER 的代理。我们在第二个模型中使用同样的优化方法，使得

这项工作做得甚至更好：将任何输入数据当作 pong，在没有其他发送数据的时候只发送一个 ping。

针对偏执海盗的信号检测

对于偏执海盗，我们选择了第二种方法。它也许不是最简单的选择：如果今天来做这个设计，我可能会尝试用乒乓的方法来代替它。然而，原理是类似的。信号检测消息异步地在两个方向流动，任意一个对等节点都可以决定另一个节点是“死的”，并停止与其交流。

在工人中，我们用如下方法来处理来自队列的检测信号：

- 计算一个*活跃度*（*liveness*），这是我们在做出队列崩溃的决定之前，允许错过的检测信号数量。它的初始值是 3，每次我们错过检测信号时，就将它递减。
- 在 zmq_poll() 循环中，每次等候 1 秒的时间，这是信号检测间隔。
- 如果在这段时间里有来自队列的任何消息，将活跃度重置为 3。
- 如果在这段时间里没有消息，减少我们的活跃度。
- 如果活跃度达到零，我们就认为队列崩溃了。
- 如果队列已经崩溃了，就销毁我们的套接字，创建一个新的套接字，并重新连接。
- 162 为了避免打开和关闭太多套接字，在重新连接之前，等待一定的时间间隔，并且每次对时间间隔加倍，直至达到 32 秒。

这就是我们如何处理队列的信号检测：

- 在发送下一个信号检测的时候执行计算，这是一个单变量，因为我们正在与一个对等节点，即队列交流。
- 在 zmq_poll() 循环中，每当经过这段时间，我们就发送一个信号检测到队列。

下面是工人的基本信号检测的代码：

```
#define HEARTBEAT_LIVENESS  3       //  3-5 是合理的
#define HEARTBEAT_INTERVAL  1000    //  毫秒
#define INTERVAL_INIT       1000    //  重新连接时间间隔的初始值
#define INTERVAL_MAX       32000    //  在指数级退避后的时间间隔

...
//  如果活跃度触及零，就认为队列已经断开连接
size_t liveness = HEARTBEAT_LIVENESS;
size_t interval = INTERVAL_INIT;

//  以常规时间间隔发出检测信号
```

```
uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

while (true) {
    zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN, 0 } };
    int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);

    if (items [0].revents & ZMQ_POLLIN) {
        //  从队列接收到任何消息
        liveness = HEARTBEAT_LIVENESS;
        interval = INTERVAL_INIT;
    }
    else
    if (--liveness == 0) {
        zclock_sleep (interval);
        if (interval < INTERVAL_MAX)
            interval *= 2;
        zsocket_destroy (ctx, worker);
        ...
        liveness = HEARTBEAT_LIVENESS;
    }
    //  如果到时间了就发送检测信号给队列
    if (zclock_time () > heartbeat_at) {
        heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        //  发送信号检测消息给队列
    }
}
```

队列做同样的工作，但它还为每名工人管理到期时间。

以下是针对你自己实现信号检测的一些技巧： 163

- 使用 zmq_poll() 或反应器作为你的应用程序的主任务的核心。
- 以建立节点之间的检测信号开始，通过模拟故障来测试，然后建立消息流的其余部分。事后添加信号检测是非常棘手的。
- 使用简单的追踪（即输出到控制台）来使这个能够工作。为了帮助你跟踪消息在节点之间的流动，使用转储的方法，如 zmsg 提供的一个功能，并且用递增的数对你的消息编号，这样你可以看到，消息间是否有空隙。
- 在实际的应用程序中，信号检测必须是可配置的，通常由对等节点双方来商定。有些节点会希望用积极的信号检测，低至 10 毫秒。有些节点则比较消极，并希望信号检测高达 30 秒。
- 如果对于不同的节点有不同的信号检测间隔，你的轮询超时时间应该是这些间隔中的最低值（最短时间）。不要使用无限超时时间。

- 用来做信号检测的套接字要和做消息传递的是同一个套接字，这样你的信号检测也可作为保持活跃的角色来防止网络连接变得陈旧（有些防火墙对沉默连接可能是不友好的）。

合同和协议

如果你很专心地阅读本章，你就会意识到，因为信号检测的原因，偏执海盗不能与简单海盗互操作。但是，我们如何定义“互操作”呢？为了保证互操作性，我们需要一种合同，一份协议，以让在不同时间和地点的不同团队编写能保证协同工作的代码，我们称之为“协议”。

没有规范的实验是可笑的，这不是真正的应用程序的合理依据。如果我们想用其他语言来编写工人怎么办？难道我们必须读取代码，来看看它是如何工作的吗？如果由于某种原因我们想要改变协议呢？如果某个协议是成功的，即使它很简单，它也会发展，并变得更加复杂。

缺乏合同是一次性应用程序的一个肯定的标志。因此，让我们为这个协议写一个合同。该怎么做呢？

在 *http://rfc.zeromq.org* 存在一个 wiki，这是我们专门作为公共 ØMQ 合同的主页而制作的。

要创建一个新的规范，注册并按照指示操作即可。这是相当简单的，但撰写技术文本不是每个人都擅长的。

我花了大约 15 分钟，以起草新的海盗模式协议（*http://rfc.zeromq.org/spec:6*）（PPP）。这不是一个大的规范，但它确实捕获足够的信息来充当争论的基础（“你的队列不是 PPP 兼容的，请修正它！”）。

164 将 PPP 转化为一个真正的协议，将需要做更多的工作：

- 在 READY 命令中，应该有一个协议版本号，以便区分不同版本的 PPP。
- 眼下，来自请求和应答的 READY 和 HEARTBEAT 都不是完全不同的。为了使它们成为不同的，我们需要包含一个“消息类型”部分的消息结构。

面向服务的可靠队列（管家模式）

进展情况的好处是，它发生的速度非常快，在律师和委员会介入前。只是几个句子前，我们还在梦想一个更好的协议，该协议将修复世界。现在，我们拥有了它：管家协议（Majordomo Protocol，MDP）（*http://rfc.zeromq.org/spec:7*）。

此单页说明书让 PPP 变得更稳定(参见图 4-4)。我们设计复杂的架构应该遵循的方法是：先编写合同，然后才编写软件来实现它们。

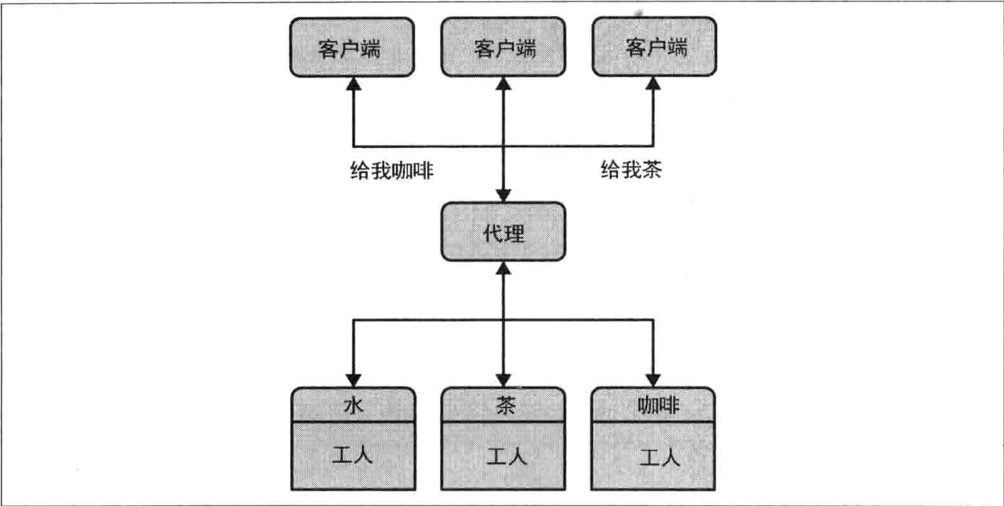


图4-4：管家模式

管家协议（MDP）用一个有趣的方式延伸并改善了 PPP：它在客户端发送的请求上增加了一个“服务名称”，并要求工人注册特定的服务。添加服务名称使得偏执海盗队列变成了面向服务的代理。MDP 的好处是，它来自于能工作的代码，一个简单的祖先协议(PPP)，以及一系列明确的改进。这使得它易于编写初稿。

为了实现管家协议，我们需要为客户端和工人编写一个框架。当应用程序开发人员只可以使用一个简单的 API 构建和测试一次时，要求他们每个人都阅读规范，并使其工作，这真的不是理智的。 165

因此，尽管我们的第一份合同（MDP 本身）定义了我们的分布式架构部件如何互相交流，第二份合同定义了用户应用程序如何跟我们要设计的技术框架交流。

管家协议分为两半，客户端一方和工人一方。由于我们将既编写客户端又编写工人应用程序，因此需要两个 API。下面是一个客户端 API 的草图，它使用一个简单的面向对象的方法：

```
mdcli_t *mdcli_new     (char *broker);
void     mdcli_destroy (mdcli_t **self_p);
zmsg_t  *mdcli_send    (mdcli_t *self, char *service, zmsg_t **request_p);
```

就是这样。我们打开一个到代理的会话，发送请求消息，得到应答，最后关闭连接。下面是一个工人 API 的草图：

```
mdwrk_t *mdwrk_new     (char *broker,char *service);
void     mdwrk_destroy (mdwrk_t **self_p);
zmsg_t  *mdwrk_recv    (mdwrk_t *self, zmsg_t *reply);
```

这或多或少是对称的，但工人对话有一点不同。一个工人第一次做 recv() 时，它传递一个空的应答。此后，它传递当前的应答，并得到一个新的请求。

客户端和工人的 API 是相当简单的，因为它们在很大程度上依赖于我们已经开发的偏执海盗代码来构造。客户端 API 如示例 4-18 所示。

示例4-18：管家客户端API（mdcliapi.c）

```
/*  =====================================================================
 *  mdcliapi.c - 管家协议客户端 API
 *  实现了位于 http://rfc.zeromq.org/spec:7 的 MDP/Worker 规范。
 *  ===================================================================== */

#include "mdcliapi.h"

//  我们的类结构
//  我们只通过类方法来访问这些属性

struct _mdcli_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    void *client;               //  到代理的套接字
    int verbose;                //  把活动输出到标准输出
    int timeout;                //  请求超时时间
    int retries;                //  请求重试次数
};
```

166

```
//  ---------------------------------------------------------------------
//  连接或重新连接到代理

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_REQ);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
```

```
        zclock_log ("I: connecting to broker at %s...", self->broker);
}
```

示例 4-19 给出了 mdcli 类的构造函数和析构函数。

示例4-19：管家客户端API（mdcliapi.c）：构造函数和析构函数

```
//  ---------------------------------------------------------------------
//  构造函数

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  毫秒
    self->retries = 3;              //  放弃前重试次数

    s_mdcli_connect_to_broker (self);
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}
```

这些都是类的方法。我们可以设置请求超时和发送请求之前重试的次数，如示例 4-20 所示。 167

示例4-20：管家客户端API（mdcliapi.c）：配置重试行为

```
//  ---------------------------------------------------------------------
//  设置请求超时时间

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}


//  ---------------------------------------------------------------------
//  设置请求重试次数

void
mdcli_set_retries (mdcli_t *self, int retries)
{
    assert (self);
    self->retries = retries;
}
```

示例 4-21 和示例 4-22 显示了 send 方法。它发送一个请求到代理并得到一个应答，即使它必须重试几次。它拥有请求消息的所有权，并在它发送完成时将其销毁。它返回应答消息，或者，如果在多次尝试后仍无应答，返回 NULL。

示例4-21：管家客户端API（mdcliapi.c）：发送请求并等待应答

```
zmsg_t *
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  用协议帧给请求加前缀
    //  第 1 帧："MDPCxy"（6 个字节，MDP/Client x.y）
    //  第 2 帧：服务名（可打印的字符串）
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    if (self->verbose) {
        zclock_log ("I: send request to '%s' service:", service);
        zmsg_dump (request);
    }
    int retries_left = self->retries;
    while (retries_left && !zctx_interrupted) {
```

```
zmsg_t *msg = zmsg_dup (request);
zmsg_send (&msg, self->client);

zmq_pollitem_t items [] = {
    { self->client, 0, ZMQ_POLLIN, 0 }
};
```

168

在任何阻塞调用中，如果发生一个错误，libzmq将返回 -1。我们在理论上可以检查不同的错误代码，但在实践中，假设它是 EINTR（Ctrl-C）即可。send 方法的主体如示例 4-22 所示。

示例4-22：管家客户端API（mdcliapi.c）：send方法的主体

```
int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
if (rc == -1)
    break;          //  中断

//  如果我们获得一个应答，就处理它
if (items [0].revents & ZMQ_POLLIN) {
    zmsg_t *msg = zmsg_recv (self->client);
    if (self->verbose) {
        zclock_log ("I: received reply:");
        zmsg_dump (msg);
    }
    //  在实际代码中，我们将更好地处理格式不好的应答
    assert (zmsg_size (msg) >= 3);

    zframe_t *header = zmsg_pop (msg);
    assert (zframe_streq (header, MDPC_CLIENT));
    zframe_destroy (&header);

    zframe_t *reply_service = zmsg_pop (msg);
    assert (zframe_streq (reply_service, service));
    zframe_destroy (&reply_service);

    zmsg_destroy (&request);
    return msg;     //  成功
}
else
if (--retries_left) {
    if (self->verbose)
        zclock_log ("W: no reply, reconnecting...");
    s_mdcli_connect_to_broker (self);
}
else {
    if (self->verbose)
```

```
                zclock_log ("W: permanent error, abandoning");
            break;          //  放弃
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing client...\n");
    zmsg_destroy (&request);
169 return NULL;
}
```

让我们来看看客户端 API 在实践中看起来的样子，带有一个执行 10 万次请求 - 应答循环的示例测试程序（参见示例 4-23）。

示例4-23：管家客户端应用程序（mdclient.c）

```
//
//  管家协议客户端示例
//  使用 mdcli API 来隐藏全部的 MDP 方面
//

//  构建这个源代码，但不创建一个库
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        zmsg_t *reply = mdcli_send (session, "echo", &request);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  中断或失败
    }
    printf ("%d requests/replies processed\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

工人 API 在示例 4-24 到示例 4-30 中描述。

示例4-24：管家工人API（mdwrkapi.c）

```
/*  =====================================================================
 *  mdwrkapi.c - 管家协议工人 API
 *  实现了位于 http://rfc.zeromq.org/spec:7 的 MDP/Worker 规范。
 *  ===================================================================== */

#include "mdwrkapi.h"

//  可靠性参数
#define HEARTBEAT_LIVENESS  3       //  3-5 是合理的
```

示例 4-25 显示的是一个工人 API 实例的结构。我们在很多 C 实例以及 CZMQ 绑定中使用一个伪面向对象的方法。

示例4-25：管家工人API（mdwrkapi.c）：工人类结构 170

```
//  我们的类结构
//  我们只通过类方法来访问这些属性

struct _mdwrk_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    char *service;
    void *worker;               //  到代理的套接字
    int verbose;                //  把活动输出到标准输出

    //  信号检测管理
    uint64_t heartbeat_at;      //  发送 HEARTBEAT 的时间
    size_t liveness;            //  剩余尝试的次数
    int heartbeat;              //  信号检测延时，以毫秒数表示
    int reconnect;              //  重新连接延时，以毫秒数表示

    int expect_reply;           //  只在开始时为零
    zframe_t *reply_to;         //  返回身份，如果有的话
};
```

我们有两个实用函数，用来将消息发送到代理和（重新）连接到代理，如示例 4-26 所示。

示例4-26：管家工人API（mdwrkapi.c）：实用函数

```
//  ---------------------------------------------------------------------
//  将消息发送到代理
//  如果没有已提供的消息，就在内部创建一个消息

static void
s_mdwrk_send_to_broker (mdwrk_t *self, char *command, char *option,
                        zmsg_t *msg)
```

```
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    // 在消息的开头叠加协议封包
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);
    zmsg_pushstr (msg, "");

    if (self->verbose) {
        zclock_log ("I: sending %s to broker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->worker);
}


171 // ---------------------------------------------------------------------
// 连接或重新连接到代理

void s_mdwrk_connect_to_broker (mdwrk_t *self)
{
    if (self->worker)
        zsocket_destroy (self->ctx, self->worker);
    self->worker = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->worker, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s...", self->broker);

    // 将服务注册到代理
    s_mdwrk_send_to_broker (self, MDPW_READY, self->service, NULL);

    // 如果活跃度触及零，就认为队列已经断开连接
    self->liveness = HEARTBEAT_LIVENESS;
    self->heartbeat_at = zclock_time () + self->heartbeat;
}
```

示例 4-27 给出了 mdwrk 类的构造函数和析构函数。

示例4-27：管家工人API（mdwrkapi.c）：构造函数和析构函数

```
// ---------------------------------------------------------------------
// 构造函数

mdwrk_t *
```

```
mdwrk_new (char *broker,char *service, int verbose)
{
    assert (broker);
    assert (service);

    mdwrk_t *self = (mdwrk_t *) zmalloc (sizeof (mdwrk_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->service = strdup (service);
    self->verbose = verbose;
    self->heartbeat = 2500;     //  毫秒
    self->reconnect = 2500;     //  毫秒

    s_mdwrk_connect_to_broker (self);
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

void
mdwrk_destroy (mdwrk_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdwrk_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self->service);
        free (self);
        *self_p = NULL;
    }
}
```

172

我们提供了两个方法来配置工人 API。你可以设置信号检测间隔和重试次数，以符合预期的网络性能（参见示例 4-28）。

示例4-28：管家工人的API（mdwrkapi.c）：配置工人

```
//  ---------------------------------------------------------------------
//  设置信号检测延时

void
mdwrk_set_heartbeat (mdwrk_t *self, int heartbeat)
{
```

```
    self->heartbeat = heartbeat;
}


//  ------------------------------------------------------------------
//  设置重新连接延时

void
mdwrk_set_reconnect (mdwrk_t *self, int reconnect)
{
    self->reconnect = reconnect;
}
```

示例 4-29 显示了 recv 方法，这个方法有点名不副实，因为它首先发送任何应答，然后等待新的请求。如果你想到一个更好的名字，请告诉我！

示例4-29：管家工人API（mdwrkapi.c）：recv方法

```
//  ------------------------------------------------------------------
//  如果有任何应答，就把它发送到代理然后等待下一个请求。

zmsg_t *
mdwrk_recv (mdwrk_t *self, zmsg_t **reply_p)
{
    //  如果我们被提供了一个应答，就格式化它并发送它
    assert (reply_p);
    zmsg_t *reply = *reply_p;
    assert (reply || !self->expect_reply);
    if (reply) {
        assert (self->reply_to);
        zmsg_wrap (reply, self->reply_to);
        s_mdwrk_send_to_broker (self, MDPW_REPLY, NULL, reply);
173     zmsg_destroy (reply_p);
    }
    self->expect_reply = 1;

    while (true) {
        zmq_pollitem_t items [] = {
            { self->worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, self->heartbeat * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->worker);
            if (!msg)
```

```
            break;          //  中断
        if (self->verbose) {
            zclock_log ("I: received message from broker:");
            zmsg_dump (msg);
        }
        self->liveness = HEARTBEAT_LIVENESS;

        //  不尝试处理错误，只是嘈杂地断言
        assert (zmsg_size (msg) >= 3);

        zframe_t *empty = zmsg_pop (msg);
        assert (zframe_streq (empty, ""));
        zframe_destroy (&empty);

        zframe_t *header = zmsg_pop (msg);
        assert (zframe_streq (header, MDPW_WORKER));
        zframe_destroy (&header);

        zframe_t *command = zmsg_pop (msg);
        if (zframe_streq (command, MDPW_REQUEST)) {
            //  我们应该弹出并保存与实际存在的一样多的地址到一个空的部分
            //  但现在，只保存一个地址……
            self->reply_to = zmsg_unwrap (msg);
            zframe_destroy (&command);
```

最后，这里是我们实际上处理一个消息的地方，如示例 4-30 所示，我们把它返回给调用者应用程序。

示例4-30：管家工人API（mdwrkapi.c）：处理消息

```
        return msg;     //  我们有一个请求要处理
        }
        else
        if (zframe_streq (command, MDPW_HEARTBEAT))
            ;               //  不对信号检测做任何事
        else
        if (zframe_streq (command, MDPW_DISCONNECT))
            s_mdwrk_connect_to_broker (self);
        else {
            zclock_log ("E: invalid input message");
            zmsg_dump (msg);
        }
        zframe_destroy (&command);
        zmsg_destroy (&msg);
    }
    else
```

174

```
        if (--self->liveness == 0) {
            if (self->verbose)
                zclock_log ("W: disconnected from broker - retrying...");
            zclock_sleep (self->reconnect);
            s_mdwrk_connect_to_broker (self);
        }
        //  如果到时间就发送 HEARTBEAT
        if (zclock_time () > self->heartbeat_at) {
            s_mdwrk_send_to_broker (self, MDPW_HEARTBEAT, NULL, NULL);
            self->heartbeat_at = zclock_time () + self->heartbeat;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing worker...\n");
    return NULL;
}
```

让我们用实现了 echo 服务的一个示例测试程序来看看工人 API 在实践中看起来的样子（参见示例 4-31）。

示例4-31：管家工人应用程序（mdworker.c）

```
//
//  管家协议工人示例
//  使用 mdwrk API 来隐藏全部的 MDP 方面
//

//  构建这个源代码，但不创建一个库
#include "mdwrkapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdwrk_t *session = mdwrk_new (
        "tcp://localhost:5555", "echo", verbose);

    zmsg_t *reply = NULL;
    while (true) {
        zmsg_t *request = mdwrk_recv (session, &reply);
        if (request == NULL)
            break;              //  工人被中断
        reply = request;        //  回应是复杂的 ... :-)
    }
    mdwrk_destroy (&session);
175 return 0;
}
```

下面是工人 API 代码的一些注意事项：

- 该API是单线程的。这意味着，例如，工人将不会在后台发送信号检测。令人高兴的是，这正是我们想要的：如果工人应用程序卡住，信号检测就会停止，而代理将停止给工人发送请求。
- 工人 API 不会执行指数退避，这个额外的复杂性是不值得的。
- 这些 API 没有做任何错误报告。如果事情与预期不同，它们就会引发一个断言（或异常，这取决于语言）。这对于一个参考实现是理想的，因此任何协议错误都将立即显示。对于实际应用程序，该 API 对抗无效的消息应该是健壮的。

你可能想知道，如果对等节点消失后又回来，ØMQ 会自动重新连接套接字，为什么工人 API 还要手动关闭它的套接字并打开一个新的套接字。回头看看简单海盗和偏执海盗工人就会理解了。如果代理崩溃后又重新回来，虽然 ØMQ 会自动重新连接工人，但对于重新将工人注册到代理，这是不够的。我知道至少有两种解决方案。最简单的方法，也是我们在这里使用的，使用信号检测为工人监视连接，如果它判断代理已崩溃，则关闭其套接字并用一个新的套接字重新开始。另一种方法是，当代理从未知的工人身上得到一个信号检测时，让代理来质询它们，并让它们重新注册。这需要协议支持。

现在让我们来设计管家代理。它的核心结构是一组队列，每个服务对应一个队列。我们将在工人出现时创建这些队列（可以在工人消失时删除它们，但现在，请先忘了这回事，因为这会使事情复杂化）。此外，我们将继续为每个服务保持一个工人队列。

代理的代码如示例 4-32 所示。

示例4-32：管家代理（mdbroker.c）

```
//
//  管家协议代理
//  一个最精简的管家协议 C 实现，这个协议的定义在
//  http://rfc.zeromq.org/spec:7 和 http://rfc.zeromq.org/spec:8 中
//
#include "czmq.h"
#include "mdp.h"

//  通常我们会从配置数据提取这些

#define HEARTBEAT_LIVENESS  3       //  3-5 是合理的
#define HEARTBEAT_INTERVAL  2500    //  毫秒
#define HEARTBEAT_EXPIRY    HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS
```
176

代理类（参见示例 4-33）定义了一个单代理实例。

示例4-33：管家代理（mdbroker.c）：代理类结构

```
typedef struct {
    zctx_t *ctx;                // 上下文
    void *socket;               // 用于客户端和工人的套接字
    int verbose;                // 把活动输出到标准输出
    char *endpoint;             // 绑定到这个端点的代理
    zhash_t *services;          // 已知的服务的散列值
    zhash_t *workers;           // 已知的工人的散列值
    zlist_t *waiting;           // 等待的工人的列表
    uint64_t heartbeat_at;      // 发送信号检测的时间
} broker_t;

static broker_t *
    s_broker_new (int verbose);
static void
    s_broker_destroy (broker_t **self_p);
static void
    s_broker_bind (broker_t *self, char *endpoint);
static void
    s_broker_worker_msg (broker_t *self, zframe_t *sender, zmsg_t *msg);
static void
    s_broker_client_msg (broker_t *self, zframe_t *sender, zmsg_t *msg);
static void
    s_broker_purge (broker_t *self);
```

服务类（参见示例 4-34）定义了一个单服务实例。

示例4-34：管家代理（mdbroker.c）：服务类结构

```
typedef struct {
    broker_t *broker;           // 代理实例
    char *name;                 // 服务名
    zlist_t *requests;          // 客户端请求的列表
    zlist_t *waiting;           // 等待的工人的列表
    size_t workers;             // 我们拥有的工人数量
} service_t;

static service_t *
    s_service_require (broker_t *self, zframe_t *service_frame);
static void
    s_service_destroy (void *argument);
static void
    s_service_dispatch (service_t *service, zmsg_t *msg);
```

177 工人类（参见示例 4-35）定义了一个工人，他是空闲或活跃的。

示例4-35：管家代理（mdbroker.c）：工人类结构

```
typedef struct {
    broker_t *broker;          //  代理实例
    char *id_string;           //  工人的身份
    zframe_t *identity;        //  用于路由的身份帧
    service_t *service;        //  如果已知，拥有服务
    int64_t expiry;            //  如果没有检测信号，工人何时过期
} worker_t;

static worker_t *
    s_worker_require (broker_t *self, zframe_t *identity);
static void
    s_worker_delete (worker_t *self, int disconnect);
static void
    s_worker_destroy (void *argument);
static void
    s_worker_send (worker_t *self, char *command, char *option,
                   zmsg_t *msg);
static void
    s_worker_waiting (worker_t *self);
```

代理的构造函数和析构函数如示例 4-36 所示。

示例4-36：管家代理（mdbroker.c）：代理的构造函数和析构函数

```
static broker_t *
s_broker_new (int verbose)
{
    broker_t *self = (broker_t *) zmalloc (sizeof (broker_t));

    //  初始化代理状态
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->verbose = verbose;
    self->services = zhash_new ();
    self->workers = zhash_new ();
    self->waiting = zlist_new ();
    self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
    return self;
}

static void
s_broker_destroy (broker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
```

```
        broker_t *self = *self_p;
        zctx_destroy (&self->ctx);
        zhash_destroy (&self->services);
        zhash_destroy (&self->workers);
        zlist_destroy (&self->waiting);
        free (self);
178     *self_p = NULL;
    }
}
```

如示例4-37所示的bind方法将该代理实例绑定到一个端点。我们可以多次调用这个方法。请注意，MDP 同时对客户端和工人使用一个套接字。

示例4-37：管家代理（mdbroker.c）：代理绑定方法

```
void
s_broker_bind (broker_t *self, char *endpoint)
{
    zsocket_bind (self->socket, endpoint);
    zclock_log ("I: MDP broker/0.2.0 is active at %s", endpoint);
}
```

示例 4-38 所示的 worker_msg 方法处理由工人发送给代理的就绪、应答、信号检测，或断开连接（READY、REPLY、HEARTBEAT、DISCONNECT）消息。

示例4-38：管家代理（mdbroker.c）：代理worker_msg方法

```
static void
s_broker_worker_msg (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 1);     //  至少有一个命令

    zframe_t *command = zmsg_pop (msg);
    char *id_string = zframe_strhex (sender);
    int worker_ready = (zhash_lookup (self->workers, id_string) != NULL);
    free (id_string);
    worker_t *worker = s_worker_require (self, sender);

    if (zframe_streq (command, MDPW_READY)) {
        if (worker_ready)               //  不是会话中的第一个命令
            s_worker_delete (worker, 1);
        else
        if (zframe_size (sender) >= 4  //  保留的服务名
        &&  memcmp (zframe_data (sender), "mmi.", 4) == 0)
            s_worker_delete (worker, 1);
        else {
            //  将工人附加到服务并将它标识为空闲
```

```
                zframe_t *service_frame = zmsg_pop (msg);
                worker->service = s_service_require (self, service_frame);
                worker->service->workers++;
                s_worker_waiting (worker);
                zframe_destroy (&service_frame);
            }
        }
        else
        if (zframe_streq (command, MDPW_REPLY)) {
            if (worker_ready) {
                //  删除并保存客户端返回的封包，                                    179
                //  并把协议标头和服务名插入进去，然后重新包装封包
                zframe_t *client = zmsg_unwrap (msg);
                zmsg_pushstr (msg, worker->service->name);
                zmsg_pushstr (msg, MDPC_CLIENT);
                zmsg_wrap (msg, client);
                zmsg_send (&msg, self->socket);
                s_worker_waiting (worker);
            }
            else
                s_worker_delete (worker, 1);
        }
        else
        if (zframe_streq (command, MDPW_HEARTBEAT)) {
            if (worker_ready)
                worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
            else
                s_worker_delete (worker, 1);
        }
        else
        if (zframe_streq (command, MDPW_DISCONNECT))
            s_worker_delete (worker, 0);
        else {
            zclock_log ("E: invalid input message");
            zmsg_dump (msg);
        }
        free (command);
        zmsg_destroy (&msg);
    }
```

示例 4-39 显示了我们如何处理一个来自客户端的请求。我们在这里直接实现管家的管理接口（MMI）请求（目前，只有 mmi.service 请求）。

示例4-39：管家代理（mdbroker.c）：代理client_msg方法

```
static void
s_broker_client_msg (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 2);      //  服务名+正文

    zframe_t *service_frame = zmsg_pop (msg);
    service_t *service = s_service_require (self, service_frame);

    //  设置把身份返回给客户端发送者的应答
    zmsg_wrap (msg, zframe_dup (sender));

    //  如果我们得到一个 MMI 服务请求，就在内部处理它
    if (zframe_size (service_frame) >= 4
    &&  memcmp (zframe_data (service_frame), "mmi.", 4) == 0) {
180     char *return_code;
        if (zframe_streq (service_frame, "mmi.service")) {
            char *name = zframe_strdup (zmsg_last (msg));
            service_t *service =
                (service_t *) zhash_lookup (self->services, name);
            return_code = service && service->workers? "200": "404";
            free (name);
        }
        else
            return_code = "501";

        zframe_reset (zmsg_last (msg), return_code, strlen (return_code));

        //  删除并保存客户端返回的封包，
        //  并把协议标头和服务名插入进去，然后重新包装封包
        zframe_t *client = zmsg_unwrap (msg);
        zmsg_push (msg, zframe_dup (service_frame));
        zmsg_pushstr (msg, MDPC_CLIENT);
        zmsg_wrap (msg, client);
        zmsg_send (&msg, self->socket);
    }
    else
        //  否则将消息分配给被请求的服务
        s_service_dispatch (service, msg);
    zframe_destroy (&service_frame);
}
```

如示例 4-40 所示，purge 方法删除那些在一段时间内没有 ping 我们的任何空闲工人。因为我们按照从最旧到最新的顺序保存工人，所以，每当发现一个活跃的工人，我们就可以停止扫描。这意味着我们将主要停留在第一个工人上，当有大量工人时，这是必不可

少的（因为我们在关键路径上调用此方法）。

示例4-40：管家代理（mdbroker.c）：代理purge方法

```
static void
s_broker_purge (broker_t *self)
{
    worker_t *worker = (worker_t *) zlist_first (self->waiting);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;                  //  工人是存活的，在这里我们已经完成
        if (self->verbose)
            zclock_log ("I: deleting expired worker: %s",
                        worker->id_string);

        s_worker_delete (worker, 0);
        worker = (worker_t *) zlist_first (self->waiting);
    }
}
```

示例 4-41 显示了在一个服务上工作的方法的实现。

示例4-41：管家代理（mdbroker.c）：服务方法 181

```
//  懒惰构造函数，它通过名字查找服务，
//  或者如果没有那个名字的服务存在，就创建一个新服务

static service_t *
s_service_require (broker_t *self, zframe_t *service_frame)
{
    assert (service_frame);
    char *name = zframe_strdup (service_frame);

    service_t *service =
        (service_t *) zhash_lookup (self->services, name);
    if (service == NULL) {
        service = (service_t *) zmalloc (sizeof (service_t));
        service->broker = self;
        service->name = name;
        service->requests = zlist_new ();
        service->waiting = zlist_new ();
        zhash_insert (self->services, name, service);
        zhash_freefn (self->services, name, s_service_destroy);
        if (self->verbose)
            zclock_log ("I: added service: %s", name);
    }
    else
```

```
        free (name);

    return service;
}

//  无论何时，当服务从 broker->services 被删除时，
//  都会自动调用服务的析构函数

static void
s_service_destroy (void *argument)
{
    service_t *service = (service_t *) argument;
    while (zlist_size (service->requests)) {
        zmsg_t *msg = zlist_pop (service->requests);
        zmsg_destroy (&msg);
    }
    zlist_destroy (&service->requests);
    zlist_destroy (&service->waiting);
    free (service->name);
    free (service);
}
```

如示例 4-42 所示，dispatch 方法将请求发送给等候的工人。

示例4-42：管家代理（mdbroker.c）：dispatch方法

```
static void
s_service_dispatch (service_t *self, zmsg_t *msg)
182 {
    assert (self);
    if (msg)                        //  对消息排队，如果有的话
        zlist_append (self->requests, msg);

    s_broker_purge (self->broker);
    while (zlist_size (self->waiting) && zlist_size (self->requests)) {
        worker_t *worker = zlist_pop (self->waiting);
        zlist_remove (self->broker->waiting, worker);
        zmsg_t *msg = zlist_pop (self->requests);
        s_worker_send (worker, MDPW_REQUEST, NULL, msg);
        zmsg_destroy (&msg);
    }
}
```

示例 4-43 显示了一个使用工人的方法的实现。

示例4-43：管家代理（mdbroker.c）：工人方法

```
//  懒惰构造函数，它通过身份查找工人，或者如果
```

```
// 没有那个身份的工人存在，就创建一个新工人

static worker_t *
s_worker_require (broker_t *self, zframe_t *identity)
{
    assert (identity);

    // self->workers 保存工人身份
    char *id_string = zframe_strhex (identity);
    worker_t *worker =
        (worker_t *) zhash_lookup (self->workers, id_string);

    if (worker == NULL) {
        worker = (worker_t *) zmalloc (sizeof (worker_t));
        worker->broker = self;
        worker->id_string = id_string;
        worker->identity = zframe_dup (identity);
        zhash_insert (self->workers, id_string, worker);
        zhash_freefn (self->workers, id_string, s_worker_destroy);
        if (self->verbose)
            zclock_log ("I: registering new worker: %s", id_string);
    }
    else
        free (id_string);
    return worker;
}

// delete 方法删除当前的工人

static void
s_worker_delete (worker_t *self, int disconnect)
{
    assert (self);
    if (disconnect)
        s_worker_send (self, MDPW_DISCONNECT, NULL, NULL);

    if (self->service) {
        zlist_remove (self->service->waiting, self);
        self->service->workers--;
    }
    zlist_remove (self->broker->waiting, self);
    // 这里隐式地调用了 s_worker_destroy
    zhash_delete (self->broker->workers, self->id_string);
}
```

183

```
//  无论何时，当工人从 broker->workers 被删除时，
//  都会自动调用工人的析构函数

static void
s_worker_destroy (void *argument)
{
    worker_t *self = (worker_t *) argument;
    zframe_destroy (&self->identity);
    free (self->id_string);
    free (self);
}
```

send 方法（参见示例 4-44）格式化一个命令并将其发送给一个工人。调用者也可以提供命令选项和消息负载。

示例4-44：管家代理（mdbroker.c）：工人send方法

```
static void
s_worker_send (worker_t *self, char *command, char *option, zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  在消息开头叠加协议封包
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);

    //  在消息开头叠加路由封包
    zmsg_wrap (msg, zframe_dup (self->identity));

    if (self->broker->verbose) {
        zclock_log ("I: sending %s to worker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->broker->socket);
}

//  这个工人现在正在等候执行工作

184
static void
s_worker_waiting (worker_t *self)
{
    //  排队到代理和服务等待列表
    assert (self->broker);
```

```
    zlist_append (self->broker->waiting, self);
    zlist_append (self->service->waiting, self);
    self->expiry = zclock_time () + HEARTBEAT_EXPIRY;
    s_service_dispatch (self->service, NULL);
}
```

最后，下面是主任务。在示例 4-45 中，我们创建一个新的代理实例，然后处理在代理套接字上的消息。

示例4-45：管家代理（mdbroker.c）：主任务

```
int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    broker_t *self = s_broker_new (verbose);
    s_broker_bind (self, "tcp://*:5555");

    //  永久地获取并处理消息，除非它被中断
    while (true) {
        zmq_pollitem_t items [] = {
            { self->socket,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        //  处理下一个输入消息，如果有的话
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->socket);
            if (!msg)
                break;          //  中断
            if (self->verbose) {
                zclock_log ("I: received message:");
                zmsg_dump (msg);
            }
            zframe_t *sender = zmsg_pop (msg);
            zframe_t *empty  = zmsg_pop (msg);
            zframe_t *header = zmsg_pop (msg);

            if (zframe_streq (header, MDPC_CLIENT))
                s_broker_client_msg (self, sender, msg);
            else
            if (zframe_streq (header, MDPW_WORKER))
                s_broker_worker_msg (self, sender, msg);
            else {
                zclock_log ("E: invalid message:");
```

185

```
                zmsg_dump (msg);
                zmsg_destroy (&msg);
            }
            zframe_destroy (&sender);
            zframe_destroy (&empty);
            zframe_destroy (&header);
        }
        //  断开连接并删除任何过期的工人
        //  如有必要，给空闲的工人发送检测信号
        if (zclock_time () > self->heartbeat_at) {
            s_broker_purge (self);
            worker_t *worker = (worker_t *) zlist_first (self->waiting);
            while (worker) {
                s_worker_send (worker, MDPW_HEARTBEAT, NULL, NULL);
                worker = (worker_t *) zlist_next (self->waiting);
            }
            self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, shutting down...\n");

    s_broker_destroy (&self);
    return 0;
}
```

这是迄今为止我们所见过的最复杂的例子。这几乎有 500 行代码，编写这个并使得它健壮一些花了我两天时间。然而，对于一个完整的面向服务的代理，这仍然是相对较短的一段代码。

这段代码的注释如下：

- 管家协议允许我们在一个套接字上处理客户端和工人。对于那些部署和管理代理的人们，这是更好的：它只位于一个 ØMQ 端点上，而不是大部分代理服务器需要的两个端点。
- 该代理正确地实现了所有 MDP/0.1（据我所知），包括如果代理发送的命令无效就断开连接、信号检测，等等。
- 它可以被扩展为运行多个线程，每个线程管理一个套接字和一组客户端和工人。这对于分解大型架构可能是有意义的。C 代码已经围绕代理类被组织起来，从而使实现这个变得微不足道。
- 一个主 / 故障转移或双活代理可靠性模型是容易的，因为代理本质上除服务存在外没有其他状态。如果它们的第一选择没有启动并运行起来的话，由客户端和工人来选择另一个代理。

- 这个示例使用 5 秒的信号检测，主要是为了减少当启用跟踪时的输出量。对于大多数局域网应用程序，实际值会比这个值低。然而，任何重试间隔时间都必须足够长，186
比如说至少 10 秒，以允许重新启动一个服务。

后来我们改进并扩展了协议和管家实现，它们现在位于自己的 GitHub 项目中。如果你想有一个正确可用的管家软件套件，请使用 GitHub 项目。

异步管家模式

上一节介绍的管家实现简单而愚蠢。客户端只是将原来的简单海盗包装在一个诱人的 API 中而已。当我在测试电脑中启动一个客户端、代理和工人时，它在 14 秒内大约可以处理 10 万个请求。这个结果部分是由于代码乐呵呵地到处复制消息帧，好像 CPU 周期全都免费而造成的。但真正的问题是，我们正在执行网络往返。虽然 ØMQ 禁用了 Nagle 算法（*http://en.wikipedia.org/wiki/Nagles_algorithm*），但往返仍然缓慢。

在理论上，理论是很好的，但在实践中，实践是更好的。让我们用一个简单的测试程序来衡量往返的实际成本。在这个测试中，我们发出了一堆消息，首先逐个等待每个消息的应答，其次作为批处理，将所有应答作为一个批次来读取。虽然这两种方法都做同样的工作，但它们给出了非常不同的结果。我们模拟了一个客户端、代理和工人。客户端的任务如示例 4-46 所示。

示例4-46：往返演示（tripping.c）

```
//
//  往返演示
//
//  虽然这个示例在单个进程中运行，
//  这只是为了易于启动和停止本示例。
//  客户端在它准备就绪时发信号给主程序。
//
#include "czmq.h"

static void
client_task (void *args, zctx_t *ctx, void *pipe)
{
    void *client = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (client, "tcp://localhost:5555");
    printf ("Setting up test...\n");
    zclock_sleep (100);

    int requests;
    int64_t start;
```

```
    printf ("Synchronous round-trip test...\n");
    start = zclock_time ();
    for (requests = 0; requests < 10000; requests++) {
        zstr_send (client, "hello");
        char *reply = zstr_recv (client);
187     free (reply);
    }
    printf ("%d calls/second\n",
        (1000 * 10000) / (int) (zclock_time () - start));

    printf ("Asynchronous round-trip test...\n");
    start = zclock_time ();
    for (requests = 0; requests < 100000; requests++)
        zstr_send (client, "hello");
    for (requests = 0; requests < 100000; requests++) {
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf ("%d calls/second\n",
        (1000 * 100000) / (int) (zclock_time () - start));
    zstr_send (pipe, "done");
}
```

这名工人的任务显示在示例 4-47 中。它所做的就是接收一条消息，并将它按原路反弹回去。

示例4-47：往返演示（tripping.c）：工人的任务

```
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");

    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

示例 4-48 显示了代理的工作。它使用 zmq_proxy() 函数在前端和后端之间切换消息。

示例4-48：往返演示（tripping.c）：代理任务

```
static void *
broker_task (void *args)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (frontend, "tcp://*:5555");
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "tcp://*:5556");
    zmq_proxy (frontend, backend, NULL);
    zctx_destroy (&ctx);
    return NULL;
}
```

188

最后，示例 4-49 给出了主任务，它启动客户端、工人和代理，然后再运行，直到客户端发信号要它停止。

示例4-49：往返演示（tripping.c）：主任务

```
int main (void)
{
    //  创建线程
    zctx_t *ctx = zctx_new ();
    void *client = zthread_fork (ctx, client_task, NULL);
    zthread_new (worker_task, NULL);
    zthread_new (broker_task, NULL);

    //  等待客户端流水线上的信号
    char *signal = zstr_recv (client);
    free (signal);

    zctx_destroy (&ctx);
    return 0;
}
```

在我的开发电脑中，运行此程序会产生如下输出：

```
Setting up test...
Synchronous round-trip test...
 9057 calls/second
Asynchronous round-trip test...
 173010 calls/second
```

这段输出的含义是：

```
设置测试……
```

```
同步往返测试……
 9057 调用 / 秒
异步往返测试……
 173010 调用 / 秒
```

请注意，客户端线程在开始之前会暂停一小段时间，这是为了避开路由器套接字的“特性”之一：如果你发送一个消息，它带有尚未连接的对等节点的地址，那么该消息会被丢弃。在这个例子中，我们不使用负载均衡机制，所以没有休眠，如果工作线程太慢，以至于连接不上，那么它就会丢失消息，使得测试变成一个烂摊子。

正如我们看到的，在最简单的情况下，往返比异步的“将它尽可能快地塞进管子里”的方法慢 20 倍。让我们看看是否能够将这个方法应用到管家模式上，以使其更快地执行。

首先，我们修改客户端 API 来在两个不同的方法中执行发送和接收：

```
mdcli_t *mdcli_new     (char *broker);
void     mdcli_destroy (mdcli_t **self_p);
int      mdcli_send    (mdcli_t *self, char *service, zmsg_t **request_p);
zmsg_t  *mdcli_recv    (mdcli_t *self);
```

189 将同步客户端 API 重构成异步的，这其实只有几分钟的工作，如示例 4-50 所示。

示例4-50：管家异步客户端API（mdcliapi2.c）

```
/*  =====================================================================
 *  mdcliapi2.c – 管家协议客户端 API
 *  实现了 http://rfc.zeromq.org/spec:7 上的 MDP/Worker 规范。
 *  ===================================================================== */

#include "mdcliapi2.h"

//  类的结构
//  我们只通过类方法访问这些属性

struct _mdcli_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    void *client;               //  到代理的套接字
    int verbose;                //  将活动输出到标准输出
    int timeout;                //  请求超时时间
};

//  ---------------------------------------------------------------------
//  连接或重新连接到代理。在这个异步类中，
//  我们用 DEALER 套接字取代 REQ 套接字，
//  这使得能够发送任意数量的请求而不必等待应答。
```

```
void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s...", self->broker);
}


//  构造函数和析构函数与 mdcliapi 中的相同，
//  但在此我们不执行重试，所以不存在重试属性。
...
...
```

不同之处在于：

- 我们用一个 DEALER 套接字来代替 REQ，所以我们使用每个请求和每个响应之前的空定界符帧来模拟 REQ。
- 我们不会重试请求，如果应用程序需要重试，那它可以自己做这件事。
- 我们将同步 send 方法分解为单独的 send 和 recv 方法。
- send 方法是异步的，并在发送后立即返回。因此调用者可以在得到一个响应之前发 190
  送许多消息。
- recv 方法等待（带有超时）响应并将其返回给调用者。

相应的客户端测试程序如示例 4-51 所示，该程序发送 10 万个消息，然后接收回 10 万个消息。

示例4-51：管家客户端应用程序（mdclient2.c）

```
//
//  管家协议客户端示例 - 异步
//  使用 mdcli API 隐藏所有的 MDP 方面
//
//  构建这个源代码，但不创建一个库
#include "mdcliapi2.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
```

```
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        mdcli_send (session, "echo", &request);
    }
    for (count = 0; count < 100000; count++) {
        zmsg_t *reply = mdcli_recv (session);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  通过 Ctrl-C 中断
    }
    printf ("%d replies received\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

代理和工人都保持不变，因为我们还没有修改协议。我们看到性能立即得到了改善。下面是同步客户端通过 100K 的请求 - 应答循环的情况：

```
$ time mdclient
100000 requests/replies processed

real    0m14.088s
user    0m1.310s
sys     0m2.670s
```

191 而下面是异步客户端，使用了一个工人：

```
$ time mdclient2
100000 replies received

real    0m8.730s
user    0m0.920s
sys     0m1.550s
```

快了两倍。还不错，但让我们发出 10 个工人，并来看看它是如何处理流量的：

```
$ time mdclient2
100000 replies received

real    0m3.863s
user    0m0.730s
sys     0m0.470s
```

因为工人严格地在上次使用的基础上获取它们的信息，所以它不是完全异步的，但使用更多的工人，它会更好地扩展。然而在我的个人电脑上，在增加到 8 名工人后，它却不

会变得更快，仅有 4 个核心的状况延绵至今。但我们只用了几分钟的工作就将吞吐量提高到了 4 倍。该代理仍是未优化的，它将大部分时间花费在复制消息帧上，而不是做零拷贝，而它本来是可以做到的。但我们利用相当低的努力，就得到了可靠的每秒 25K 的请求 - 应答调用。

然而，异步管家模式并非完美。它有一个根本弱点，即如果不付出更多的工作，它无法在代理崩溃时存活。如果查看 mdcliapi2 代码，你会看到它并没有试图在失败后重新连接。一个适当的重新连接需要执行以下操作：

- 每个请求有一个编号而每个应答都有一个匹配的编号，这需要修改协议以强制执行。
- 跟踪并保存客户端 API 中的所有未完成的请求（即那些尚未收到应答的）。
- 在故障转移的情况下，让客户端 API 将所有未完成的请求重新发送到代理。

这不是一个败招，但它确实表明，性能往往意味着复杂性。对于管家这是值得做的吗？这取决于你的用例。对于每个会话调用一次名称查找服务，这不值得。但对于向成千上万客户端提供服务的 Web 前端，或许是值得的。

服务发现

所以，我们有一个很好的面向服务的代理，但我们没有办法知道一个特定的服务是否可用。我们知道一个请求何时失败，但不知道它为什么失败。要能问代理“echo 服务是否在运行？”这样的问题将是有益的，实现这个功能最明显的方法，就是修改我们的 MDP/Client 通信协议来添加提出这种问题的命令。但 MDP/Client 具有简单的巨大魅力。192
添加服务发现将使它与 MDP/Worker 协议一样复杂。

另一种选择是，做电子邮件服务器所做的工作，并要求无法投递的请求被退回。在一个异步的世界中，这可以很好地工作，但同时也增加了复杂性。我们需要能够将返回的请求与应答区分开来的方法，并妥善处理这些问题。

让我们来尝试利用我们已经建成的东西，即在 MDP 之上构建，而不是修改它。服务发现本身就是一个服务。这可能确实是几个管理服务之一，如“禁用服务 X”，“提供统计信息”等。我们要的是：不影响协议的或现有的应用程序的可扩展的通用解决方案。

有一个在 MDP 之上分层的小型 RFC，即：管家管理接口（MMI）（*http://rfc.zeromq.org/spec:8*）。我们已经在代理中实现了它，但除非你阅读了整个代码，否则你可能错过了它。我将解释它是如何在代理中工作的。

- 当客户端请求了以 mmi. 开头的服务时，我们不把这个请求路由到一个工人，而是在内部处理它。

- 我们在代理中只处理一个服务，这个服务是 `mmi.service`，即服务发现服务。
- 请求的有效载荷是一个外部服务的名称（一个实际的名称，由工人提供）。
- 该代理将返回“200”（OK）或“404”（未找到），这取决于是否有工人注册那个服务。

示例 4-52 显示了如何在应用程序中使用服务发现。

示例4-52：**通过管家的服务发现**（mmiecho.c）

```
//
//  MMI 回应查询示例
//

//  构建这个源代码，但不创建一个库
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  这是我们要查找的服务
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");

    //  这是我们要向其发送请求的服务
193 zmsg_t *reply = mdcli_send (session, "mmi.service", &request);

    if (reply) {
        char *reply_code = zframe_strdup (zmsg_first (reply));
        printf ("Lookup echo service: %s\n", reply_code);
        free (reply_code);
        zmsg_destroy (&reply);
    }
    else
        printf ("E: no response from broker, make sure it’s running\n");

    mdcli_destroy (&session);
    return 0;
}
```

分别在有和没有工人正在运行时试着运行这个程序，你应该看到这个小程序相应地报告“200”或“404”。

示例代理中的MMI实现是靠不住的。例如，如果某个工人消失了，服务仍然“存在”。在实践中，代理应该删除那些在某个可配置的超时时间后仍没有任何工人的服务。

幂等服务

幂等性不是你服用的某种药片。它的意思是，重复某个操作是安全的。检查时钟是幂等的，将某人的信用卡借给某人的孩子则不是幂等的。虽然许多客户端到服务器的用例都是幂等的，但有些用例不是。幂等用例的例子包括：

- 无状态的任务分配，也就是说，一条流水线，其中服务器是完全基于一个请求中提供的状态来计算一个应答的无状态工人。在这种情况下，多次执行同样的请求是安全的（虽然效率很低）。
- 名称服务，它将逻辑地址转换为要绑定或连接到的节点。在这种情况下，多次做出同样的查找请求是安全的。

而下面是非幂等用例的例子：

- 日志服务。人们不希望同样的日志信息被多次记录。
- 对下游节点有影响（例如，给其他节点发送信息）的任何服务。如果该服务多次得到了同样的请求，下游节点就会得到重复的信息。
- 以某种非幂等方式修改共享数据的任何服务，例如，借记银行账户的服务绝对不是幂等的。

当我们的服务器应用程序不是幂等的时候，我们必须更仔细地考虑，它们何时可能会崩 194
溃。如果有个应用程序在它闲置或处理请求的时候崩溃了，这通常是没关系的。我们可以使用数据库事务来确保一个借方和一个贷方总是一起记录完成，如果存在的话。然而，如果服务器在发送应答时死机，这是一个问题，因为就它而言，它已经完成了它的工作。

如果在应答返回给客户端的途中，网络崩溃了，会出现同样的问题。客户端会认为服务器死机了，而将重新发送请求，服务器会将同样的工作做两次，这不是我们希望的。

为了处理非幂等操作，我们使用检测和拒绝重复请求的相当标准的解决方案。这意味着：

- 客户端必须为每个请求加盖一个独特的客户端标识符和一个唯一的消息编号。
- 服务器在发回一个应答之前，使用客户端 ID 和消息编号的组合作为键来存储它。
- 服务器从给定的客户端获取请求时，首先检查它是否拥有该客户端 ID 和消息编号的应答。如果有这样的应答，它就不处理该请求，而只是重新发送应答。

断开连接的可靠性（泰坦尼克模式）

一旦你意识到管家是一个“可靠的”消息代理，你也许会添加一些旋转铁锈（即铁基硬盘盘片）。毕竟，这适用于所有的企业消息传递系统。它是一个如此诱人的想法，以至于当必须面对它的负面时有点令人伤感。但残酷的犬儒主义是我的特色之一。所以，不

想把旋转铁锈型代理置于架构中心的部分原因是：

- 如你所见，懒惰海盗客户端执行得非常好。它可以跨越架构的整个范围工作，从直接的客户端到服务器到分布式队列代理。它确实倾向于认为工人是无状态的和幂等的，但我们可以解决这个限制，而不诉诸铁锈。
- 铁锈带来了一系列的问题，从性能下降到额外的部件，你必须管理、维修它们，并且当它们在日常业务开始时不可避免地遭到损坏的时候，还要在早上 6 点处理这类恐慌。海盗模式的总体优点是其简单性。它们不会崩溃。如果还是担心硬件，你可以转到完全没有代理的一个对等模式（我将在本章的后面解释）。

话虽如此，然而，基于铁锈的可靠性有一个合理的用例，这个用例是一个异步的断开连
195 接的网络。它解决了使用海盗模式的一个主要问题，即一个客户端必须实时地等待回答。如果客户端和工人只是偶尔连接的（想想电子消息的例子），我们就不能使用客户端和工人之间的一个无状态的网络。我们必须把状态放在中间。

所以，下面提供了泰坦尼克模式（参见图 4-5），在此模式中，我们将消息写入磁盘，以确保它们不会丢失，无论客户端和工人连接是多么零散。正如我们在服务发现中所做的，我们将在 MDP 之上放一层泰坦尼克，而不是扩展 MDP。这是奇妙的懒惰，因为这意味着我们可以在一个专门的工人，而不是代理中实现我们的即发即忘（fire-and-forget）的可靠性。它的好处有下面几个原因：

- 这容易得多，因为我们分而治之：代理处理消息路由，而工人处理可靠性。
- 它允许我们将用一种语言编写的代理与用另一种语言编写的工人混用。
- 它允许我们独立地开发即发即忘技术。

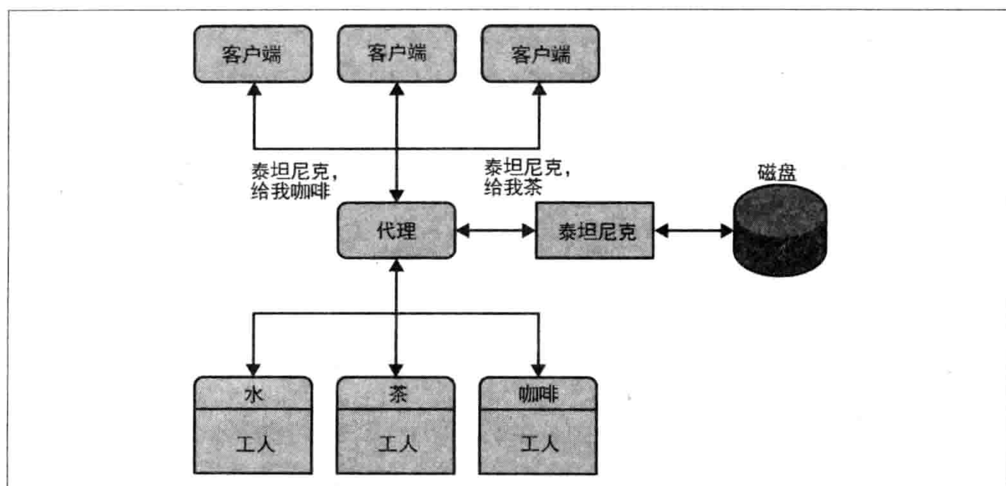


图4-5：泰坦尼克模式

这种模式唯一的缺点是，代理和硬盘之间有一个额外的网络跃点。但与其优点比较，这个缺点是很值得付出的。

有许多方法都可以建立一个持久的请求 - 应答架构。我们的目标是一个简单、无痛的方法。我能想出的最简单的设计，是一个“代理服务”，这个花了我几个小时来研究，那即是，泰坦尼克完全不影响工人。如果客户端希望立即收到应答，它就直接与服务交流，并希望该服务可用。如果客户端乐意等待一段时间，它会改成与泰坦尼克交流并且询问它：“嘿，哥们，你会在我去买东西的时候帮我照顾这个吗？”

泰坦尼克因此既是工人又是客户端。客户端和泰坦尼克之间的对话，如下所示： 196

- 客户端：“请接受我的这个请求。”泰坦尼克：“好的，完成了。”
- 客户端：“你有一个应答要给我吗？”泰坦尼克：“是的，在这儿呢。”（或者，“不，还没有。”）
- 客户端：“好了，你现在可以擦除这一请求，我很高兴。”泰坦尼克：“好的，完成了。”

而泰坦尼克与代理和工人之间的对话是这样的：

- 泰坦尼克：“嘿，代理，有一个咖啡服务吗？”
- 代理：“嗯，是啊，好像有。”
- 泰坦尼克：“嘿，咖啡服务，请帮我处理这个问题。”
- 咖啡：“当然可以，给你。”
- 泰坦尼克：“香极了！”

你可以通过这些和可能的失败场景工作。如果一个工人在处理一个请求时死掉，泰坦尼克就会无限期地重试。如果某个应答在某处丢失了，泰坦尼克也将重试。如果请求得到处理，但客户端没有得到应答，它会再次询问。如果泰坦尼克在处理请求或应答时崩溃，那么客户端将重试。只要请求被完全提交到了安全的存储中，工作就不会丢失。

握手是迂腐的，但可以被管道化，也就是说，客户端可以使用异步管家模式做很多工作，而在之后得到回应。

我们需要用某种方式为客户端请求它的应答。我们有很多客户端要求同一个服务，并且客户端可能会消失，并以不同的身份重新出现。下面是一个简单且安全合理的解决方案：

- 每个请求都生成一个通用唯一识别码（UUID），泰坦尼克将请求排队后，将这个 UUID 返回给客户端。
- 当客户端请求一个应答时，它必须指定原始请求的 UUID。

在实际情况下，客户端想要将其请求的 UUID 安全地存储起来，如存在本地数据库中。

在我们深入下去写另一个正式的规范前，让我们考虑客户端如何与泰坦尼克交流。一种方法是使用单个服务并发送给它三个不同的请求类型。另一种方式，这似乎是更简单的，就是用三个服务：

`titanic.request`
: 存储请求消息，并返回一个这个请求的 UUID。

`titanic.reply`
: 对于给定请求的 UUID，如果它有应答的话，就获取该应答。

197 `titanic.close`
: 确认某个应答已被存储和处理。

我们将只做一个多线程的工人，正如我们已经从使用 ØMQ 多线程的经验所看到的，这是微不足道的。不过，让我们先勾勒出泰坦尼克从 ØMQ 消息和帧的角度所看到的总体结构。这给了我们泰坦尼克服务协议（TSP）（*http://rfc.zeromq.org/spec:9*）。

客户端应用程序使用 TSP 而不是通过 MDP 直接访问一个服务，这显然要做更多的工作。最简短的健壮的“回应”客户端的例子如示例 4-53 所示。

示例4-53：泰坦尼克客户端的例子（ticlient.c）

```
//
//  泰坦尼克客户端示例
//  实现 http://rfc.zeromq.org/spec:9 的客户端

//  构建这个源代码，但不创建一个库
#include "mdcliapi.c"

//  调用 TSP 服务
//  若成功则返回响应（状态码 200 OK），否则返回 NULL
//
static zmsg_t *
s_service_call (mdcli_t *session, char *service, zmsg_t **request_p)
{
    zmsg_t *reply = mdcli_send (session, service, request_p);
    if (reply) {
        zframe_t *status = zmsg_pop (reply);
        if (zframe_streq (status, "200")) {
            zframe_destroy (&status);
            return reply;
        }
```

```
        else
        if (zframe_streq (status, "400")) {
            printf ("E: client fatal error, aborting\n");
            exit (EXIT_FAILURE);
        }
        else
        if (zframe_streq (status, "500")) {
            printf ("E: server fatal error, aborting\n");
            exit (EXIT_FAILURE);
        }
    }
    else
        exit (EXIT_SUCCESS);    //  中断或失败

    zmsg_destroy (&reply);
    return NULL;                //  不成功，但不关心其原因
}
```

主任务（参见示例 4-54）通过发送一个回应请求来测试我们的服务调用。

示例4-54：泰坦尼克客户端的例子（ticlient.c）：主任务 198

```
int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  1. 向 Titanic 发出 'echo' 请求
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = s_service_call (
        session, "titanic.request", &request);

    zframe_t *uuid = NULL;
    if (reply) {
        uuid = zmsg_pop (reply);
        zmsg_destroy (&reply);
        zframe_print (uuid, "I: request UUID");
    }

    //  2. 等待直到我们得到一个应答
    while (!zctx_interrupted) {
        zclock_sleep (100);
        request = zmsg_new ();
        zmsg_add (request, zframe_dup (uuid));
        zmsg_t *reply = s_service_call (
```

```
            session, "titanic.reply", &request);

        if (reply) {
            char *reply_string = zframe_strdup (zmsg_last (reply));
            printf ("Reply: %s\n", reply_string);
            free (reply_string);
            zmsg_destroy (&reply);

            //  3. 关闭请求
            request = zmsg_new ();
            zmsg_add (request, zframe_dup (uuid));
            reply = s_service_call (session, "titanic.close", &request);
            zmsg_destroy (&reply);
            break;
        }
        else {
            printf ("I: no reply yet, trying again...\n");
            zclock_sleep (5000);     //  在 5 秒后重试
        }
    }
    zframe_destroy (&uuid);
    mdcli_destroy (&session);
    return 0;
}
```

199 当然，这可以是也应该是包装在某种框架或 API 中的。要求一般的应用程序开发人员学习消息传递的全部细节，这是不健康的：它伤害了他们的大脑，花费了时间，并提供了太多途径来引入容易出问题的复杂性。此外，它使得添加智能很困难。

例如，此客户端阻塞在每个请求上，而在实际应用中，我们会希望在任务执行时做有用功。它需要用一些不平凡的流水线来建立一个后台线程并干净地与之交流。这是你想要在一个不错的简单的 API 中包装的那类东西，它使得一般开发人员也不至于误用。这是我们曾用于管家模式的相同方法。

泰坦尼克实现如示例 4-55 到示例 4-60 所示。按照建议，该服务器使用三个线程处理三种服务。它使用尽可能最粗暴的方式：每个消息一个文件地完全持久化到磁盘。它是如此简单，它也使人担心。唯一复杂的部分是，为了避免反复读取目录，它将所有的请求保持在一个单独的队列中。

示例4-55：泰坦尼克代理的例子（titanic.c）

```
//
//  泰坦尼克服务
//
```

```
//  实现http://rfc.zeromq.org/spec:9的服务器端

//  构建这个源代码，但不创建一个库
#include "mdwrkapi.c"
#include "mdcliapi.c"

#include "zfile.h"
#include <uuid/uuid.h>

//  返回形式为可打印字符串的一个新UUID
//  调用者在处理完返回的字符串后必须释放它

static char *
s_generate_uuid (void)
{
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    uuid_generate (uuid);
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuid [byte_nbr] >> 4];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuid [byte_nbr] & 15];
    }
    return uuidstr;
}

//  返回新分配的用于给定的UUID的请求文件名

#define TITANIC_DIR ".titanic"

static char *
s_request_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.req", uuid);
    return filename;
}

//  返回新分配的用于给定的UUID的应答文件名

static char *
s_reply_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.rep", uuid);
    return filename;
}
```

200

titanic.request 任务（参见示例 4-56）等待针对该服务的请求。它把每个请求写入到磁盘，并返回一个 UUID 给客户端。客户端使用 titanic.reply 服务异步地捡起应答。

示例4-56：泰坦尼克代理的例子（titanic.c）：泰坦尼克请求服务

```
static void
titanic_request (void *args, zctx_t *ctx, void *pipe)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.request", 0);
    zmsg_t *reply = NULL;

    while (true) {
        //  如果应答不为空，就发送它
        //  并且随后从代理获取下一个请求
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  中断，退出

        //  确保消息目录存在
        zfile_mkdir (TITANIC_DIR);

        //  生成 UUID 并将消息保存到磁盘
        char *uuid = s_generate_uuid ();
        char *filename = s_request_filename (uuid);
        FILE *file = fopen (filename, "w");
        assert (file);
        zmsg_save (request, file);
        fclose (file);
        free (filename);
        zmsg_destroy (&request);

201     //  将 UUID 发送到消息队列
        reply = zmsg_new ();
        zmsg_addstr (reply, uuid);
        zmsg_send (&reply, pipe);

        //  将 UUID 发送回客户端
        //  这通过在循环开头的 mdwrk_recv() 完成
        reply = zmsg_new ();
        zmsg_addstr (reply, "200");
        zmsg_addstr (reply, uuid);
        free (uuid);
    }
    mdwrk_destroy (&worker);
}
```

titanic.reply 任务如示例 4-57 所示，检查是否有指定请求的应答（通过 UUID），并相应地返回 200（OK）、300（待定）或 400（未知）。

示例4-57：泰坦尼克代理的例子（titanic.c）：泰坦尼克回复服务

```
static void *
titanic_reply (void *context)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.reply", 0);
    zmsg_t *reply = NULL;

    while (true) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  中断，退出

        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
        if (zfile_exists (rep_filename)) {
            FILE *file = fopen (rep_filename, "r");
            assert (file);
            reply = zmsg_load (NULL, file);
            zmsg_pushstr (reply, "200"):     // OK
            fclose (file);
        }
        else {
            reply = zmsg_new ();
            if (zfile_exists (req_filename))
                zmsg_pushstr (reply, "300"); // 待定
            else
                zmsg_pushstr (reply, "400"); // 未知
        }
        zmsg_destroy (&request);
        free (uuid);
        free (req_filename);
        free (rep_filename);
    }
    mdwrk_destroy (&worker);
    return 0;
}
```

202

titanic.close 任务如示例 4-58 所示，删除任何等待应答的请求（通过 UUID 指定）。它是幂等的，所以多次调用它是安全的。

示例4-58：泰坦尼克代理的例子（titanic.c）：泰坦尼克关闭任务

```
titanic_close (void *context)
static void *
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.close", 0);
    zmsg_t *reply = NULL;

    while (true) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  中断，退出

        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
        zfile_delete (req_filename);
        zfile_delete (rep_filename);
        free (uuid);
        free (req_filename);
        free (rep_filename);

        zmsg_destroy (&request);
        reply = zmsg_new ();
        zmsg_addstr (reply, "200");
    }
    mdwrk_destroy (&worker);
    return 0;
}
```

示例 4-59 显示了泰坦尼克工人的主线程。它启动三个子线程，用于请求、应答和关闭服务。然后，它会使用简单的蛮力磁盘队列将请求调度给工人。它从 `titanic.request` 服务接收各个请求的 UUID，将这些保存到磁盘文件，然后将每个请求抛给 MDP 工人，直到得到一个响应。

示例4-59：泰坦尼克代理的例子（titanic.c）：工人的任务

```
static int s_service_success (char *uuid);
```

203

```
int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    zctx_t *ctx = zctx_new ();

    void *request_pipe = zthread_fork (ctx, titanic_request, NULL);
```

```
zthread_new (titanic_reply, NULL);
zthread_new (titanic_close, NULL);

//  主调度器循环
while (true) {
    //  如果没有活动，我们将每秒调度一次
    zmq_pollitem_t items [] = { { request_pipe, 0, ZMQ_POLLIN, 0 } };
    int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
    if (rc == -1)
        break;              //  中断
    if (items [0].revents & ZMQ_POLLIN) {
        //  确保消息目录存在
        zfile_mkdir (TITANIC_DIR);

        //  将 UUID 追加到队列中，用前缀 "-" 表示待定
        zmsg_t *msg = zmsg_recv (request_pipe);
        if (!msg)
            break;          //  中断
        FILE *file = fopen (TITANIC_DIR "/queue", "a");
        char *uuid = zmsg_popstr (msg);
        fprintf (file, "-%s\n", uuid);
        fclose (file);
        free (uuid);
        zmsg_destroy (&msg);
    }
    //  粗暴的调度器
    char entry [] = "?.......:.......:.......:.......:";
    FILE *file = fopen (TITANIC_DIR "/queue", "r+");
    while (file && fread (entry, 33, 1, file) == 1) {
        //  如果还在等待，用 "-" 作为 UUID 前缀
        if (entry [0] == '-') {
            if (verbose)
                printf ("I: processing request %s\n", entry + 1);
            if (s_service_success (entry + 1)) {
                //  将队列条目标记为已处理
                fseek (file, -33, SEEK_CUR);
                fwrite ("+", 1, 1, file);
                fseek (file, 32, SEEK_CUR);
            }
        }
        //  跳过行结束符，LF 或 CRLF
        if (fgetc (file) == '\r')
            fgetc (file);
        if (zctx_interrupted)
            break;
```

```
        }
        if (file)
            fclose (file);
    }
204     return 0;
}
```

在代理代码的最后一部分（参见示例 4-60），我们首先使用 MMI 查找管家代理来检查所请求的 MDP 服务是否都被定义了。如果服务存在，我们发送请求并使用传统的 MDP 客户端 API 等待一个应答。这并不意味着快速，只是很简单而已。

示例4-60：泰坦尼克代理的例子（titanic.c）：尝试调用一个服务

```
static int
s_service_success (char *uuid)
{
    //  加载请求的消息，服务将是第一帧
    char *filename = s_request_filename (uuid);
    FILE *file = fopen (filename, "r");
    free (filename);

    //  如果客户端已经关闭请求，则视为成功
    if (!file)
        return 1;

    zmsg_t *request = zmsg_load (NULL, file);
    fclose (file);
    zframe_t *service = zmsg_pop (request);
    char *service_name = zframe_strdup (service);

    //  用短的超时时间创建 MDP 客户端会话
    mdcli_t *client = mdcli_new ("tcp://localhost:5555", FALSE);
    mdcli_set_timeout (client, 1000);  //  1 秒
    mdcli_set_retries (client, 1);     //  只重试 1 次

    //  用 MMI 协议来检查服务是否可用
    zmsg_t *mmi_request = zmsg_new ();
    zmsg_add (mmi_request, service);
    zmsg_t *mmi_reply = mdcli_send (client, "mmi.service", &mmi_request);
    int service_ok = (mmi_reply
        && zframe_streq (zmsg_first (mmi_reply), "200"));
    zmsg_destroy (&mmi_reply);

    int result = 0;
    if (service_ok) {
        zmsg_t *reply = mdcli_send (client, service_name, &request);
```

```
        if (reply) {
            filename = s_reply_filename (uuid);
            FILE *file = fopen (filename, "w");
            assert (file);
            zmsg_save (reply, file);
            fclose (file);
            free (filename);
            result = 1;
        }
        zmsg_destroy (&reply);
    }
    else
        zmsg_destroy (&request);

    mdcli_destroy (&client);
    free (service_name);
    return result;
}
```

205

为了验证这一点，启动 mdbroker 和 titanic，然后运行 ticlient。现在随意启动 mdworker，你应该看到客户端得到了响应并愉快地退出。

关于这段代码的一些注意事项：

- 注意，某些循环通过发送消息启动，而另一些循环通过接收消息启动。这是因为泰坦尼克充当不同角色，它既是一个客户端又是一个工人。
- 泰坦尼克代理使用 MMI 服务发现协议来只给看上去正在运行的服务发送请求。由于我们的小型管家代理中的 MMI 实现是相当差的，所以它将无法在所有的时间工作。
- 我们使用 inproc 连接将来自 titanic.request 服务的新请求数据发送到主调度器。这使得调度器不必扫描磁盘目录，加载所有的请求文件，并按照日期 / 时间对其进行排序。

这个例子的重点不在于它的性能（虽然我没有测试过，但它肯定是差得可怕的），而在于它如何良好地实现了可靠性的合同。要试用一下这个例子，先启动 *mdbroker* 和 *titanic* 程序。然后启动 *ticlient*，再启动 *mdworker* echo 服务。你可以在运行所有这 4 个部件时使用 *-v* 选项来执行详细活动跟踪。除了客户端，可以停止并重新启动任何一个部件，什么都不会丢失。

如果想在实际工作中使用泰坦尼克模式，你很快就会问："我们如何使它运行得更快？"下面就是我想要做的，从示例实现开始：

- 使用单个磁盘文件，而不是多个文件来保存所有数据。通常，操作系统能更好地处理少数几个大文件而不是许多较小的文件。
- 将磁盘文件组织为一个循环缓冲区，以便新的请求可以连续写入（带有非常偶然的回绕）。一个全速写入磁盘文件的线程，能迅速地开展工作。
- 在内存中保持索引，并在启动时从磁盘缓存重建索引。这样可以节省为保持索引在
206 磁盘上完全安全而需要额外的磁头快速运动。你会希望在每个消息后 `fsync` 一次，或者，如果你愿意在系统万一出现故障时丢失最后 M 个消息，也可以每 N 毫秒 `fsync` 一次。
- 使用固态硬盘，而不是旋转磁盘。
- 预分配整个文件，或者将它分配为大块，这使得循环缓冲区能够根据需要增长和收缩。这样做避免了碎片，并确保大部分读取和写入都是连续的。

等等。我倒是不建议将消息存储在数据库中，甚至不建议保存在一个“快速的”键 / 值存储中，除非你真的喜欢一个特定的数据库，并且不担心性能。因为你会为抽象付出高昂的代价——花费相当于原始磁盘文件 10 至 1000 倍的时间。

如果你想使泰坦尼克模式*更可靠*，可以将请求重复地发到第二台服务器，并将其放置在离你的主要位置足够远的第二个位置，这个足够远只是指它可以在主要位置遭到核袭击时生存，但并不是要远得产生太长的延时。

如果你想使泰坦尼克模式*更快但不那么可靠*，那么可以在内存中存储请求和应答。这会给你提供一个断开网络连接的功能，但当泰坦尼克服务器本身崩溃时，请求将无法生存。

高可用性对（双星模式）

双星模式（Binary Star Pattern）配置两台服务器作为主 / 备用高可用性对（参见图 4-6）。在任何给定的时间内，其中一台（活动服务器）接受来自客户端应用程序的连接。另一台（被动服务器）不执行任何操作，但两台服务器相互监视。如果活动的那台服务器从网络消失，经过一定时间，被动的服务器就接管工作，充当活动服务器。

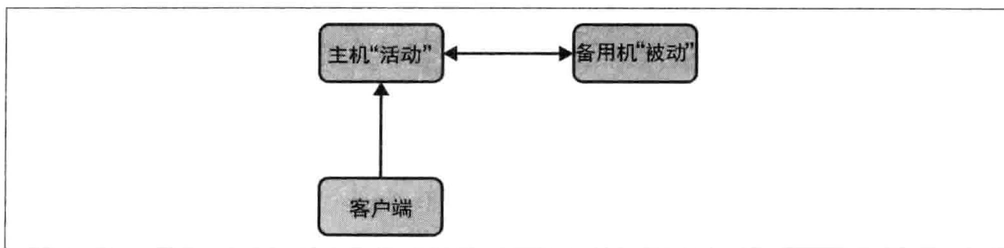


图4-6：高可用性对，正常运行

我们在 iMatix 为我们的 OpenAMQ 服务器（*http://www.Openamp.org*）开发了双星模式。我们将它设计为：

- 提供一个简单的高可用性解决方案。
- 足够简单，以便真正理解和使用。
- 在需要并且仅在需要时进行可靠的故障转移。 207

假设我们有一个双星对正在运行，这里有会导致故障转移的不同情况（参见图 4-7）：

- 运行在主服务器中的硬件发生一个致命的问题（电源爆炸、机器着火，或者干脆有人误拔了电线），并且消失。应用程序看到这一点，并重新连接到备用服务器。
- 主服务器所处的网段崩溃——也许是因为瞬间电流过大烧毁了路由器——而应用程序开始重新连接到备用服务器。
- 主服务器崩溃或被操作者清除，并且不会自动重新启动。

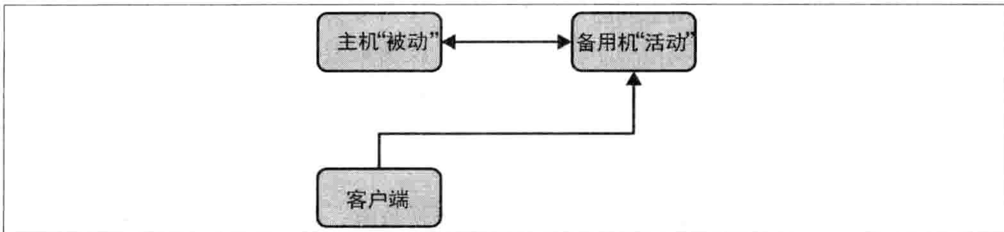


图4-7：故障转移过程中的高可用性对

从故障转移恢复的工作原理如下：

- 操作者修复导致主服务器从网络上消失的任何问题，并重新启动它。
- 操作者在备用服务器会对应用程序造成最小中断的时候，立刻停止它。
- 当应用程序已重新连接到主服务器时，操作者重新启动备用服务器。

恢复（使用主服务器充当活动服务器）是手动操作。惨痛的经验告诉我们，自动恢复是不可取的。有下面几个原因：

- 故障转移造成对应用程序的服务中断，这可能会持续 10 ~ 30 秒。如果有一个真正的紧急情况，这比全部故障好得多。但是，如果恢复会造成一个进一步的此类故障，这种情况最好是避开高峰，即当用户已经离开了网络时再发生。
- 当出现紧急情况时，绝对的首要任务是确定那些试图解决的事情。自动恢复会为系统管理员带来不确定性，它不能在不重复检查的情况下确定哪些是负责的服务器。 208

- 自动恢复可能造成这样的情况，其中网络故障转移然后恢复，操作者将在分析发生了什么事时面临困境。服务的中断是存在的，但原因不清楚。

话虽如此，如果主服务器正在运行（再次）且备份服务器失败，双星模式将自动故障转移回到主服务器。事实上，这就是我们发起恢复的方法。

双星对的关机过程是执行以下操作之一：

1. 停止被动服务器，然后在以后的任何时间停止活动服务器。
2. 按任意顺序停止两台服务器，但彼此在几秒钟之内停止。

停止活动服务器，然后以任何长于故障转移超时时间的延迟停止被动服务器将导致应用程序断开连接，然后重新连接，然后再断开连接，这可能会干扰用户。

详细需求

双星模式是尽可能简单的，但它同时还能准确地工作。事实上，目前的设计是第三个完整的重新设计。每当我们发现前一个设计太复杂，试图做太多工作时，我们就去掉一些功能，直到得到一个可以理解、易于使用，并且足够可靠、值得使用的设计。

下面这些都是高可用性架构的需求：

- 故障转移是为了针对灾难性的系统故障，如硬件故障、火灾、事故等提供保险。还有更简单的方法可以从普通的服务器崩溃中恢复，而且我们已经涵盖了这些方法。
- 故障转移时间应小于 60 秒，最好在 10 秒以内。
- 故障转移必须自动发生，而恢复必须手动进行。我们希望应用程序自动切换到备份服务器，但是，除非操作者已经修复存在的任何问题并且决定，这是很好的再次中断应用程序的时机，否则我们不希望它们切换回主服务器。
- 客户端应用程序的语义应该是简单并易于开发人员理解的。理想的情况下，它们应该被隐藏在客户端 API 中。
- 209 网络架构师应该有明确的指示来规定如何避免可能导致“脑裂综合征”的设计，即在一个双星对中的两台服务器都认为自己是活动服务器。
- 这两台服务器启动的顺序应该没有依赖关系。
- 必须能够在不停止客户端应用程序（尽管它们可能会被迫重新连接）的情况下按计划停止并重新启动任一台服务器。
- 操作者必须能够在任何时候都可以同时监视两台服务器。
- 必须能够使用高速专用网络连接来连接两台服务器。即，故障转移同步必须能够使用特定的 IP 路由。

我们做如下假设：

- 单个备用服务器提供了足够的保险，我们并不需要多个级别的备用。
- 主服务器和备用服务器都有同样的工作能力来承载应用负载。我们并不试图跨越服务器平衡负载。
- 有足够的预算来支付一台几乎在所有时间里什么也不做的完全冗余的备用服务器。

我们并不试图涵盖以下内容：

- 使用活动的备用服务器或负载均衡。在一个双星对中，备用服务器处于非活动状态，并且不做有用的工作，直到主服务器处于脱机状态为止。
- 以任何方式对持久性消息或事务的处理。我们假设存在不可靠的(也可能是不可信的)服务器或双星对网络。
- 对网络的任何自动探索。双星对是在网络中手动和明确定义的，并对应用程序是已知的（至少在它们的配置数据中）。
- 服务器之间的状态或消息复制。所有服务器端的状态失败后必须由应用程序重新创建它们。

下面是我们在双星模式中使用的关键术语：

主机

这是正常或初始的活动服务器。

备用机

这通常是被动的服务器。如果主服务器从网络中消失，并当客户端应用程序要求与备份服务器进行连接的时候，它会变成活动的。

活动 210

接受客户端连接的服务器。至多有一个活动的服务器。

被动

如果活动服务器消失，负责接管的服务器。请注意，当一个双星对正常运行时，主服务器处于活动状态，并且备用机是被动的。当故障转移发生时，这二者的角色切换。

要配置一个双星对，需要：

1. 把备份服务器的位置告诉主服务器。
2. 把主服务器的位置告诉备份服务器。
3. 可选的，调整故障转移的响应时间，对于两台服务器，这个时间必须是相同的。

值得关注的主要调整是，你希望服务器多久检查一次它们的对等状态，以及你想要多快地启动故障转移。在我们的例子中，故障转移的超时默认值为 2000 毫秒。如果你减少这个值，备份服务器将更快地接管为活动服务器，但可能在主服务器可以恢复的情况下接管。例如，你可能已经在一个 shell 脚本中将主服务器包装了，如果它崩溃了，就用这个脚本重新启动它。在这种情况下，超时时间应该比以重新启动主服务器所需要的时间更长。

对于客户端应用程序，为了与一个双星对正确地配合工作，它们必须：

1. 知道这两台服务器的地址。
2. 尝试连接到主服务器，如果失败了，再连接到备用服务器。
3. 检测失败的连接，通常采用信号检测的方式。
4. 尝试重新连接到主服务器，然后连接到备份服务器（按顺序），其中，重试之间的延迟应该至少与服务器故障转移超时时间一样长。
5. 重新创建所有需要的状态。
6. 重新发送故障转移过程中丢失的消息，如果消息必须可靠。

这不是简单的工作，而我们通常会把这个包装在对真正的最终用户应用程序隐藏它的 API 中。

下面这些都是双星模式的主要限制：

- 一个服务器进程不能是多个双星对的一部分。
- 主服务器可以有且只有一台备用服务器。
- 被动服务器没有做有用的工作，因此被“浪费”了。
- 备用服务器必须能够处理全部的应用程序负载。
- 211 故障转移配置不能在运行时修改。
- 客户端应用程序必须做一些工作，才能从故障转移中受益。

避免脑裂症状

当一个集群的不同部分在同一时间都认为自己是活动的时候，就发生了*脑裂症状*（*split-brain syndrome*）。它会导致应用程序互相无视对方。双星模式有一个用于检测和消除脑裂的算法，这是基于一个三方决策机制（一台服务器，除非它获取了应用程序的连接请求，并且不能看到它的对等服务器，它才会决定要成为活动的）得出的。

然而，（误）设计愚弄这个算法的网络仍然是可能的。一个典型的场景是在两栋楼之间分布的一个双星对，其中每栋楼也有一组应用程序，并在两栋楼之间有单一的网络链路。破坏这个链路将创建两组客户端应用程序，每组都使用双星对的一半，并且每台故障转

移服务器都会变成活动的。

为了防止脑裂情况，我们必须使用专用的网络链路连接双星对，这个专用的网络链路可以与将它们两者插入到相同的交换机一样简单，或者比这更好的，直接采用两台机器之间的交叉电缆。

我们绝对不能将一个双星体系结构分裂成两个孤岛，而每个孤岛各带有一组应用程序。虽然这可能是常见的网络架构类型，但在这种情况下，我们应该用联合，而不是高可用性故障转移。

适当偏执的网络配置将使用两个，而不是单个专用集群互联。进一步地讲，应要求用于集群的网卡和用于消息通信业务的网卡是不同的，甚至可能位于服务器硬件的不同的PCI路径上。我们的目标是将网络中可能出现的故障与集群中可能出现的故障分离。网络端口具有相对较高的故障率。

双星实现

事不宜迟，下面是双星服务器的概念验证实现，从示例 4-61 开始。主服务器和备份服务器运行相同的代码，而它们的角色由调用者选择。

示例4-61：双星服务器（bstarsrv.c）

```
//
//  双星服务器的概念验证实现。这台服务器不做
//  实际工作，它只是演示双星故障转移模式

#include "czmq.h"

//  任何时点我们可能所处的状态
typedef enum {
    STATE_PRIMARY = 1,          //  主服务器，等待节点连接
    STATE_BACKUP = 2,           //  备用服务器，等待节点连接
    STATE_ACTIVE = 3,           //  活动的 – 接受连接
    STATE_PASSIVE = 4           //  被动的 – 不接受连接
} state_t;

//  从我们的节点可能所处的状态开始的事件
typedef enum {                                                    212
    PEER_PRIMARY = 1,           //  HA 节点是待定主服务器
    PEER_BACKUP = 2,            //  HA 节点是待定备用服务器
    PEER_ACTIVE = 3,            //  HA 节点是活动的
    PEER_PASSIVE = 4,           //  HA 节点是被动的
    CLIENT_REQUEST = 5          //  客户端发出请求
} event_t;
```

```
//  有限状态机
typedef struct {
    state_t state;              //  当前状态
    event_t event;              //  当前事件
    int64_t peer_expiry;        //  节点被认为“死机”时
} bstar_t;

//  发送状态信息的频度
//  如果节点在两个信号检测中无回应，那么它“死机”了
#define HEARTBEAT 1000          //  以毫秒计
```

双星设计的核心是它的有限状态机（FSM）。FSM 在一个时间执行一个事件。我们将一个事件应用到当前的状态，它会检查该事件是否被接受，并且如果是这样，就设定一个新的状态（参见示例 4-62）。

示例4-62：双星服务器（bstarsrv.c）：双星状态机

```
static Bool
s_state_machine (bstar_t *fsm)
{
    Bool exception = FALSE;

    //  这些是 PRIMARY 和 BACKUP 状态，我们等待变成
    //  ACTIVE 或 PASSIVE，取决于我们从对等节点获得的事件
    if (fsm->state == STATE_PRIMARY) {
        if (fsm->event == PEER_BACKUP) {
            printf ("I: connected to backup (passive), ready as active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: connected to backup (active), ready as passive\n");
            fsm->state = STATE_PASSIVE;
        }
        //  接受客户端连接
    }
    else
    if (fsm->state == STATE_BACKUP) {
213     if (fsm->event == PEER_ACTIVE) {
            printf ("I: connected to primary (active), ready as passive\n");
            fsm->state = STATE_PASSIVE;
        }
        else
        //  当充当备用服务器时，拒绝客户端连接
        if (fsm->event == CLIENT_REQUEST)
```

```
            exception = TRUE;
    }
    else
```

活动（ACTIVE）和被动（PASSIVE）状态都在示例 4-63 中列出。

示例4-63：双星服务器（bstarsrv.c）：活动和被动状态

```
    if (fsm->state == STATE_ACTIVE) {
        if (fsm->event == PEER_ACTIVE) {
            //  两台活动的服务器意味着脑裂
            printf ("E: fatal error - dual actives, aborting\n");
            exception = TRUE;
        }
    }
    else
    //  服务器是被动的
    //  如果节点看上去宕机了，CLIENT_REQUEST 事件会触发故障转移
    if (fsm->state == STATE_PASSIVE) {
        if (fsm->event == PEER_PRIMARY) {
            //  节点正在重新启动 – 变得活动，节点将成为被动的
            printf ("I: primary (passive) is restarting, ready as active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_BACKUP) {
            //  节点正在重新启动 – 变得活动，节点将成为被动的
            printf ("I: backup (passive) is restarting, ready as active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_PASSIVE) {
            //  两个被动将意味着集群无响应
            printf ("E: fatal error - dual passives, aborting\n");
            exception = TRUE;
        }
        else
        if (fsm->event == CLIENT_REQUEST) {
            //  如果已经过了超时时间，节点就变成活动的
            //  这是触发故障转移的客户端请求
            assert (fsm->peer_expiry > 0);
            if (zclock_time () >= fsm->peer_expiry) {
                //  如果节点死机了，切换到活动状态
                printf ("I: failover successful, ready as active\n");
                fsm->state = STATE_ACTIVE;
            }
            else
```

214

```
                // 如果节点正在运行，那么拒绝连接
                exception = TRUE;
        }
    }
    return exception;
}
```

示例 4-64 显示了我们的主任务。首先，将套接字绑定 / 连接到我们的节点，并确保会得到正确的状态消息。我们使用三个套接字：一个用来发布状态，一个用来订阅状态，还有一个用于客户端请求 / 回复。

示例4-64：双星服务器（bstarsrv.c）：主任务

```
int main (int argc, char *argv [])
{
    // 参数可以是如下之一：
    //      -p  主服务器，位于 tcp://localhost:5001
    //      -b  备用服务器，位于 tcp://localhost:5002
    zctx_t *ctx = zctx_new ();
    void *statepub = zsocket_new (ctx, ZMQ_PUB);
    void *statesub = zsocket_new (ctx, ZMQ_SUB);
    zsockopt_set_subscribe (statesub, "");
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    bstar_t fsm = { 0 };

    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: Primary active, waiting for backup (passive)\n");
        zsocket_bind (frontend, "tcp://*:5001");
        zsocket_bind (statepub, "tcp://*:5003");
        zsocket_connect (statesub, "tcp://localhost:5004");
        fsm.state = STATE_PRIMARY;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: Backup passive, waiting for primary (active)\n");
        zsocket_bind (frontend, "tcp://*:5002");
        zsocket_bind (statepub, "tcp://*:5004");
        zsocket_connect (statesub, "tcp://localhost:5003");
        fsm.state = STATE_BACKUP;
    }
    else {
        printf ("Usage: bstarsrv { -p | -b }\n");
        zctx_destroy (&ctx);
        exit (0);
    }
```

我们现在处理在两个输入套接字上的事件，并通过有限状态机（参见示例 4-65）一次性处理这些事件。我们对客户端请求所做的工作是简单地将它回应回去。

示例4-65：双星服务器（bstarsrv.c）：处理套接字输入 215

```
// 为下一个输出的状态消息设置计时器
int64_t send_state_at = zclock_time () + HEARTBEAT;
while (!zctx_interrupted) {
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { statesub, 0, ZMQ_POLLIN, 0 }
    };
    int time_left = (int) ((send_state_at - zclock_time ()));
    if (time_left < 0)
        time_left = 0;
    int rc = zmq_poll (items, 2, time_left * ZMQ_POLL_MSEC);
    if (rc == -1)
        break;              // 上下文已被关闭

    if (items [0].revents & ZMQ_POLLIN) {
        // 有一个客户端请求
        zmsg_t *msg = zmsg_recv (frontend);
        fsm.event = CLIENT_REQUEST;
        if (s_state_machine (&fsm) == FALSE)
            // 通过把请求回应回去答复客户端
            zmsg_send (&msg, frontend);
        else
            zmsg_destroy (&msg);
    }
    if (items [1].revents & ZMQ_POLLIN) {
        // 有来自我们的节点的状态，作为一个事件执行
        char *message = zstr_recv (statesub);
        fsm.event = atoi (message);
        free (message);
        if (s_state_machine (&fsm))
            break;          // 出错，因此退出
        fsm.peer_expiry = zclock_time () + 2 * HEARTBEAT;
    }
    // 如果超时，就把状态发送给节点
    if (zclock_time () >= send_state_at) {
        char message [2];
        sprintf (message, "%d", fsm.state);
        zstr_send (statepub, message);
        send_state_at = zclock_time () + HEARTBEAT;
    }
}
```

```
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    //  关闭套接字和上下文
    zctx_destroy (&ctx);
    return 0;
}
```

216 现在让我们来看看客户端的代码，它从示例 4-66 开始。

示例4-66：双星客户端（bstarcli.c）

```
//
//  双星客户端概念验证实现。
//  这个客户端不做任何实际工作，它只是用来演示双星故障转移模型。

#include "czmq.h"

#define REQUEST_TIMEOUT     1000    //  毫秒
#define SETTLE_DELAY        2000    //  故障转移前的延时

int main (void)
{
    zctx_t *ctx = zctx_new ();

    char *server [] = { "tcp://localhost:5001", "tcp://localhost:5002" };
    uint server_nbr = 0;

    printf ("I: connecting to server at %s...\n", server [server_nbr]);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, server [server_nbr]);

    int sequence = 0;
    while (!zctx_interrupted) {
        //  我们发出一个请求，然后工作以获得一个应答
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            //  以超时时间来轮询套接字以获取一个应答
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  中断
```

我们在客户端使用一个懒惰海盗策略。如果在超时时间内没有应答，就关闭套接字，然后再试一次，如示例 4-67 所示。在双星模式中，客户端投票决定哪台服务器是主服务器，因此，客户端必须尝试轮流连接到每台服务器。

示例4-67：双星客户端（bstarcli.c）：客户端的主体

```
            if (items [0].revents & ZMQ_POLLIN) {
                //  从服务器获得一个应答，必须匹配序号
                char *reply = zstr_recv (client);
                if (atoi (reply) == sequence) {
                    printf ("I: server replied OK (%s)\n", reply);
                    expect_reply = 0;
                    sleep (1);  //  每秒一个请求
                }
                else
                    printf ("E: bad reply from server: %s\n", reply);
                free (reply);
            }
            else {
                printf ("W: no response from server, failing over\n");

                //  旧套接字是易混淆的，关闭它并打开一个新的套接字
                zsocket_destroy (ctx, client);
                server_nbr = (server_nbr + 1) % 2;
                zclock_sleep (SETTLE_DELAY);
                printf ("I: connecting to server at %s...\n",
                        server [server_nbr]);
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, server [server_nbr]);

                //  再次在新套接字上发送请求
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

217

为了测试我们的双星实现，以任何顺序启动服务器和客户端：

```
bstarsrv -p     # Start primary
bstarsrv -b     # Start backup
bstarcli
```

然后，你可以通过清除主服务器发起故障转移，并通过重新启动主服务器并清除备用机来恢复。注意客户端投票是如何触发故障转移和恢复的。

双星由一个有限状态机驱动（参见图 4-8）。白色状态接受客户端请求，而灰色状态拒绝它们。事件是对等状态，因此“对等活动”是指其他服务器已经告诉我们它是活动的。“客户端请求”表示我们已经收到客户端请求。“客户端投票”表示我们已经收到客户端请求并且我们的节点对于两个信号检测一直处于非活动状态。

图4-8：双星有限状态机

218 请注意，该服务器使用 PUB-SUB 套接字来进行状态交换。在这里，其他的套接字组合都不能工作。如果没有节点准备好接收一个消息，PUSH 和 DEALER 就会阻塞。如果节点消失后又回来，那么 PAIR 不重新连接。ROUTER 需要对等节点的地址，才可以向它发送一个消息。

双星反应器

双星模式是有用和通用的，足以打包成一个可重复使用的反应器类。该反应器然后运行，并当它有一个消息要处理时调用我们的代码。这比将双星代码复制 / 粘贴到我们想要那种功能的每台服务器要好得多。

在 C 语言中，我们包装了之前看到的 CZMQ `zloop` 类。`zloop` 让你注册处理程序，对套接字和计时器事件做出反应。在双星反应器中，我们提供投票者和状态变化（活动到被动，

反之亦然）的处理程序。下面是 bstar API：

```
// 创建一个新双星实例，使用本地（绑定）和
// 远程（连接）端点来设置对等服务器
bstar_t *bstar_new (int primary, char *local, char *remote);

// 销毁一个双星实例
void bstar_destroy (bstar_t **self_p);

// 返回底层 zloop 反应器，用于计时器和阅读器
// 的注册和取消
zloop_t *bstar_zloop (bstar_t *self);

// 注册投票阅读器
int bstar_voter (bstar_t *self, char *endpoint, int type,
                 zloop_fn handler, void *arg);

// 注册主状态变化处理程序
void bstar_new_active (bstar_t *self, zloop_fn handler, void *arg);
void bstar_new_passive (bstar_t *self, zloop_fn handler, void *arg);

// 启动反应器，如果一个回调函数返回 -1，
// 或者进程接收到 SIGINT 或 SIGTERM 信号，反应器就结束
int bstar_start (bstar_t *self);
```

219

这个类的实现如示例 4-68 所示。

示例4-68：双星核心类（bstar.c）

```
/* =====================================================================
 * bstar - 双星反应器
 * ===================================================================== */

#include "bstar.h"

// 任何时点我们可能所处的状态
typedef enum {
    STATE_PRIMARY = 1,          // 主服务器，等待节点连接
    STATE_BACKUP = 2,           // 备用服务器，等待节点连接
    STATE_ACTIVE = 3,           // 活动的，接受连接
    STATE_PASSIVE = 4           // 被动的，不接受连接
} state_t;

// 从我们的节点可能所处的状态开始的事件
typedef enum {
    PEER_PRIMARY = 1,           // HA 节点是待定主服务器
    PEER_BACKUP = 2,            // HA 节点是待定备用服务器
```

```
    PEER_ACTIVE = 3,              //  HA 节点是活动的
    PEER_PASSIVE = 4,             //  HA 节点是被动的
    CLIENT_REQUEST = 5            //  客户端发出请求
} event_t;

//  类的结构

struct _bstar_t {
    zctx_t *ctx;                  //  专用上下文
    zloop_t *loop;                //  反应循环
    void *statepub;               //  状态发布者
    void *statesub;               //  状态订阅者
    state_t state;                //  当前状态
    event_t event;                //  当前事件
    int64_t peer_expiry;          //  节点被认为"死机"时
    zloop_fn *voter_fn;           //  投票套接字处理程序
    void *voter_arg;              //  投票处理程序的参数
    zloop_fn *active_fn;          //  在变得活动时调用
    void *active_arg;             //  处理程序的参数
    zloop_fn *passive_fn;         //  在变成被动时调用
    void *passive_arg;            //  处理程序的参数
};

    //  有限状态机与概念验证服务器中的相同。
    //  要详细地理解这个反应器，请首先阅读 CZMQ zloop 类。
...
```

220

示例 4-69 包含了 bstar 类的构造函数。必须告诉它我们是一个主服务器还是备用服务器，并且提供绑定和连接到的本地和远程端点。

示例4-69：双星核心类（bstar.c）：构造函数

```
bstar_t *
bstar_new (int primary, char *local, char *remote)
{
    bstar_t
        *self;

    self = (bstar_t *) zmalloc (sizeof (bstar_t));

    //  初始化双星
    self->ctx = zctx_new ();
    self->loop = zloop_new ();
    self->state = primary? STATE_PRIMARY: STATE_BACKUP;

    //  为去往对等节点的状态创建发布者
```

```
    self->statepub = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->statepub, local);

    //  为来自对等节点的状态创建订阅者
    self->statesub = zsocket_new (self->ctx, ZMQ_SUB);
    zsockopt_set_subscribe (self->statesub, "");
    zsocket_connect (self->statesub, remote);

    //  设置基本反应器事件
    zloop_timer (self->loop, BSTAR_HEARTBEAT, 0, s_send_state, self);
    zmq_pollitem_t poller = { self->statesub, 0, ZMQ_POLLIN };
    zloop_poller (self->loop, &poller, s_recv_state, self);
    return self;
}
```

析构函数（参见示例 4-70）关闭 bstar 反应器。

示例4-70：双星核心类（bstar.c）：析构函数

```
void
bstar_destroy (bstar_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        bstar_t *self = *self_p;
        zloop_destroy (&self->loop);
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}
```

221

该 zloop 方法（参见示例 4-71）返回底层 zloop 反应器，因此我们可以添加额外的计时器和阅读器。

示例4-71：双星核心类（bstar.c）：zloop方法

```
zloop_t *
bstar_zloop (bstar_t *self)
{
    return self->loop;
}
```

示例 4-72 所示的 voter 方法注册一个客户端投票器套接字。在此套接字上接收到的消息为双星 FSM 提供 CLIENT_REQUEST 事件，并被传递到提供的应用程序处理程序中。我们要求每个 bstar 实例只有一个投票器。

示例4-72：双星核心类（bstar.c）：投票方法

```
int
bstar_voter (bstar_t *self, char *endpoint, int type, zloop_fn handler,
             void *arg)
{
    //  保存实际的处理程序 + 参数，以便以后调用它
    void *socket = zsocket_new (self->ctx, type);
    zsocket_bind (socket, endpoint);
    assert (!self->voter_fn);
    self->voter_fn = handler;
    self->voter_arg = arg;
    zmq_pollitem_t poller = { socket, 0, ZMQ_POLLIN };
    return zloop_poller (self->loop, &poller, s_voter_ready, self);
}
```

接下来，我们注册处理程序，每当有状态变化时就会调用它们。

示例4-73：双星核心类（bstar.c）：注册状态变化的处理程序

```
void
bstar_new_active (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->active_fn);
    self->active_fn = handler;
    self->active_arg = arg;
}

void
222 bstar_new_passive (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->passive_fn);
    self->passive_fn = handler;
    self->passive_arg = arg;
}
```

然后我们启用 / 禁用详细的跟踪，用于调试（参见示例 4-74）。

示例4-74：双星核心类（bstar.c）：启用/禁用跟踪

```
void bstar_set_verbose (bstar_t *self, Bool verbose)
{
    zloop_set_verbose (self->loop, verbose);
}
```

最后，我们开始配置反应器（参见示例 4-75）。如果任何处理程序返回 -1 给反应器，或者如果进程接收到 SIGINT 或 SIGTERM 信号，反应器就将结束。

示例4-75：双星核心类（bstar.c）：启动反应器

```
int
bstar_start (bstar_t *self)
{
    assert (self->voter_fn);
    s_update_peer_expiry (self);
    return zloop_start (self->loop);
}
```

这为我们提供了示例 4-76 所示的服务器的简短主程序。

示例4-76：双星服务器，使用核心类（bstarsrv2.c）

```
//
//  双星服务器，使用 bstar 反应器
//

//  构建这个源代码，但不创建一个库
#include "bstar.c"

//  回应服务
int s_echo (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    zmsg_t *msg = zmsg_recv (poller->socket);
    zmsg_send (&msg, poller->socket);
    return 0;
}

int main (int argc, char *argv [])
{
    //  参数可以是如下之一：
    //      -p  主服务器，位于 tcp://localhost:5001
    //      -b  备用服务器，位于 tcp://localhost:5002
    bstar_t *bstar;                                                223
    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: Primary active, waiting for backup (passive)\n");
        bstar = bstar_new (BSTAR_PRIMARY,
            "tcp://*:5003", "tcp://localhost:5004");
        bstar_voter (bstar, "tcp://*:5001", ZMQ_ROUTER, s_echo, NULL);
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: Backup passive, waiting for primary (active)\n");
        bstar = bstar_new (BSTAR_BACKUP,
            "tcp://*:5004", "tcp://localhost:5003");
        bstar_voter (bstar, "tcp://*:5002", ZMQ_ROUTER, s_echo, NULL);
```

```
    }
    else {
        printf ("Usage: bstarsrvs { -p | -b }\n");
        exit (0);
    }
    bstar_start (bstar);
    bstar_destroy (&bstar);
    return 0;
}
```

无代理可靠性（自由职业者模式）

当我们经常将 ØMQ 解释为“无代理消息传递”时，却关注这么多以代理为基础的可靠性，这似乎具有讽刺意味。不过，如同在现实生活中那样，在消息传递中，中间人既有负担，也有好处。在实践中，大多数的消息架构受益于分布式和代理的消息传递的组合。在你可以自由地决定想要做什么取舍时，你会得到最好的结果。这就是为什么我可以开车 20 分钟到一个批发商那里买五箱酒用于聚会，但我也可以步行 10 分钟到一个街头小店买一瓶酒用于晚餐的原因。我们对上下文高度敏感的相对时间、精力和成本估值对现实世界的经济是必不可少的。它们对于一个以信息为基础的最佳架构也是必不可少的。

这就是为什么 ØMQ 虽然不强加给你一个以代理为中心的架构，但它确实给你提供了构建代理，又名代理服务器（*proxy*）的工具（而且，到目前为止，我们已经建立了十几个不同的代理，只是为了练习）。

因此，我们将通过解构我们迄今为止已经建成的以代理为基础的可靠性，并把它放回一个我称之为自由职业者模式的分布式对等架构来结束这一章。我们的用例将是一个名称解析服务。这是使用 ØMQ 架构的一个常见问题：我们怎么知道要连接到哪个终端呢？在代码中硬编码 TCP/IP 地址是非常脆弱的。使用配置文件则会产生一个管理噩梦。设想一下，如果要把“google.com”识别为“74.125.230.82”，你不得不在你使用的每台电脑或手机上手工配置 Web 浏览器的情况。

一个 ØMQ 名称服务（我们将制作一个简单的实现）必须做到以下几点：

224 • 将一个逻辑名称解析为至少一个绑定端点和一个连接端点。一个现实的名称服务将提供多个绑定端点，也可能提供多个连接端点。
• 允许我们管理多个并行环境（例如，“测试”与“生产”环境），而无须修改代码。
• 可靠，因为如果它不可用，应用程序就不能连接到网络。

从一些观点来看，把一个名称服务放在面向服务的管家代理的背后，是聪明的。然而，只是将名称服务作为客户端可以直接连接到的服务器对外公开，这么做更简单，也更不

值得大惊小怪。如果我们正确地这样做了，名称服务就会成为我们需要在代码或配置文件中硬编码的唯一的全局网络端点。

我们准备处理的故障类型是服务器崩溃并重新启动、服务器忙于循环、服务器过载和网络问题。为了获得可靠性，我们将创建名称服务器池，因此如果一台服务器崩溃或消失，客户端就可以连接到另一台，依此类推。在实践中，两台名称服务器就足够了，但在这个例子中，我们假定服务器池可以是任意大小的（参见图 4-9）。

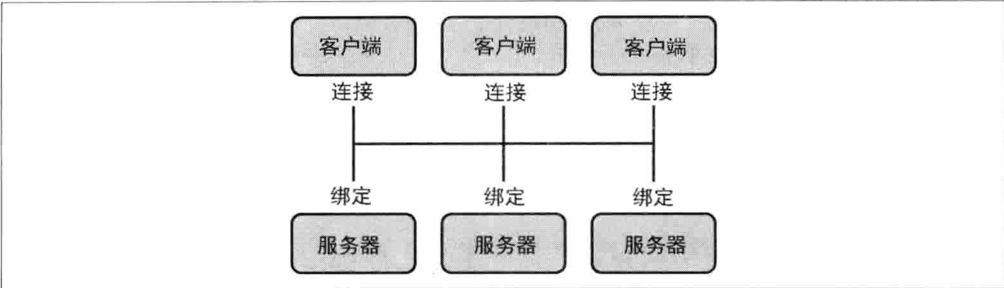


图4-9：自由职业者模式

在这种架构中，很多客户端直接连接到一小部分服务器。该服务器绑定到各自的地址。这与类似管家模式的以代理为基础的方法有根本上的不同，在那些方法中，工人连接到代理。客户端有几个选项：

- 使用 REQ 套接字和懒惰海盗模式。这个方法容易，但需要一些额外的智能，以便客户端不至于愚蠢地一遍又一遍地尝试重新连接到死机的服务器。
- 利用 DEALER 套接字并发出请求（这将负载均衡到所有连接的服务器），直到它们得到一个应答为止。这个方法有效，但不优雅。
- 使用 ROUTER 套接字，以便客户端可以针对特定的服务器。但是，客户端怎么才能 225
知道服务器套接字的身份呢？要么服务器必须首先 ping 客户端（复杂），要么服务器必须使用一个硬编码的客户端已知的固定身份（讨厌）。

我们将在下面各小节实现所有这些方法。

模型一：简单的重试和故障转移

所以，我们的菜单提供以下选择：简单的、粗暴的、复杂的，或者讨厌的。让我们先从简单的模型开始，然后再制作出纠结的模型。我们将采用懒惰海盗模式，并重写它，以使之与多台服务器端点配合工作。我们将首先启动一台或多台服务器，指定一个绑定的端点作为参数（参见示例 4-77）。

示例4-77：自由职业者服务器，模型一（flserver1.c）

```
//
// 自由职业者服务器 - 模型一
// 简单的回应服务
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: echo service is ready at %s\n", argv [1]);
    while (true) {
        zmsg_t *msg = zmsg_recv (server);
        if (!msg)
            break;          // 中断
        zmsg_send (&msg, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

然后，我们将启动客户端（参见示例 4-78），指定一个或多个连接端点作为参数。

226 示例4-78：自由职业者客户端，模型一（flclient1.c）

```
//
// 自由职业者客户端 - 模型一
// 使用 REQ 套接字来查询一个或多个服务
//
#include "czmq.h"

#define REQUEST_TIMEOUT     1000
#define MAX_RETRIES         3       // 放弃前的最大重试次数

static zmsg_t *
s_try_request (zctx_t *ctx, char *endpoint, zmsg_t *request)
```

```
{
    printf ("I: trying echo service at %s...\n", endpoint);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, endpoint);

    //  发送请求，安全地等待应答
    zmsg_t *msg = zmsg_dup (request);
    zmsg_send (&msg, client);
    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
    zmsg_t *reply = NULL;
    if (items [0].revents & ZMQ_POLLIN)
        reply = zmsg_recv (client);

    //  在任何情况下关闭套接字，我们已完成它的处理
    zsocket_destroy (ctx, client);
    return reply;
}
```

如果客户端只与一台服务器交流，它使用懒惰海盗策略。如果它有两台或更多的服务器需要交流，它会对每台服务器都只尝试一次。主要的客户端任务如示例 4-79 所示。

示例4-79：自由职业者客户端，模型一（flclient1.c）：客户端任务

```
int main (int argc, char *argv [])
{
    zctx_t *ctx = zctx_new ();
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = NULL;

    int endpoints = argc - 1;
    if (endpoints == 0)
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
    else
    if (endpoints == 1) {
        //  对于一个端点，重试 N 次
        int retries;
        for (retries = 0; retries < MAX_RETRIES; retries++) {
            char *endpoint = argv [1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  成功
            printf ("W: no response from %s, retrying...\n", endpoint);
        }
    }
```

227

```
    else {
        //  对于多个端点，每个端点最多尝试一次
        int endpoint_nbr;
        for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_nbr++) {
            char *endpoint = argv [endpoint_nbr + 1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  成功
            printf ("W: no response from %s\n", endpoint);
        }
    }
    if (reply)
        printf ("Service is running OK\n");

    zmsg_destroy (&request);
    zmsg_destroy (&reply);
    zctx_destroy (&ctx);
    return 0;
}
```

一个示例运行的命令如下：

```
flserver1 tcp://*:5555 &
flserver1 tcp://*:5556 &
flclient1 tcp://localhost:5555 tcp://localhost:5556
```

虽然基本的方法是懒惰海盗模式，但客户端的目的是只得到一个成功的回复。它有两种技术，这取决于我们正在运行的是一台服务器还是多台服务器：

- 如果使用一台服务器，客户端将重试几次，与懒惰海盗模式完全一样。
- 如果使用多台服务器，客户端将对每台服务器最多尝试一次，直到它收到应答或已经尝试了所有的服务器。

这解决了懒惰海盗模式的主要弱点，即它不能切换到备用或后备服务器。

然而，这样的设计在实际应用程序中不会很好地工作。如果我们连接了许多套接字，并且主名称服务器宕机，我们将每次都遇到这种痛苦的超时时间。

228

模型二：粗暴猎枪屠杀

让我们使用一个 DEALER 套接字切换客户端。我们的目标是，无论某个特定的服务器是在运行还是宕机，都要确保在最短的时间内获得一个返回的应答。我们的客户端采用这种方法：

- 设置某些东西，再连接到所有服务器。
- 当有一个请求时，有多少服务器就发射多少次请求。
- 等待第一个应答，并获取它。
- 忽略任何其他应答。

在实践中会出现的情况是，当所有服务器都在运行时，ØMQ 将会分发请求，让每台服务器获取一个请求，并发送一个应答。当有任何服务器处于脱机状态并断开时，ØMQ 将把请求分发到其他服务器。所以，在某些情况下，服务器可能会多次得到相同的请求。

对于客户端，更使人烦恼的是，我们会得到多个返回的应答，但不能保证我们会得到应答的准确数量。请求和应答可能会丢失（例如，如果服务器在处理一个请求时崩溃）。

因此，我们必须对请求编号，并忽略任何与请求编号不符合的应答。虽然我们的模型一服务器会工作，因为它是一个回应服务器，但巧合不是理解问题的一个好的基础，所以我们将在这里制作一个模型二服务器，它处理了该消息，并返回带有正确编号且内容为“OK”的应答。我们将使用的消息由两部分组成：一个序号和一个正文。

我们将首先启动一台或多台服务器，并且每次都指定绑定的端点，如示例 4-80 所示。

示例4-80：自由职业者服务器，模型二（flserver2.c）

```
//
//  自由职业者服务器 - 模型二
//  执行某些工作，用 OK 和消息序号来应答
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: service is ready at %s\n", argv [1]);
    while (true) {
        zmsg_t *request = zmsg_recv (server);
        if (!request)
            break;          //  中断
        //  如果对错误的客户端运行，则讨厌地失败
        assert (zmsg_size (request) == 2);
```

229

```
        zframe_t *identity = zmsg_pop (request);
        zmsg_destroy (&request);

        zmsg_t *reply = zmsg_new ();
        zmsg_add (reply, identity);
        zmsg_addstr (reply, "OK");
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

然后我们将启动客户端，指定连接端点作为参数，如示例 4-81 所示。

示例4-81：自由职业者客户端，模型二（flclient2.c）

```
//
//  自由职业者客户端 - 模型二
//  使用 DEALER 套接字来发射一个或多个服务
//
#include "czmq.h"

//  将客户端 API 设计为一个类，使用 CZMQ 风格
#ifdef __cplusplus
extern "C" {
#endif

typedef struct _flclient_t flclient_t;
flclient_t *flclient_new (void);
void        flclient_destroy (flclient_t **self_p);
void        flclient_connect (flclient_t *self, char *endpoint);
zmsg_t     *flclient_request (flclient_t *self, zmsg_t **request_p);

#ifdef __cplusplus
}
#endif

//  如果没有一个服务在这段时间内应答，则放弃
#define GLOBAL_TIMEOUT 2500

int main (int argc, char *argv [])
230 {
    if (argc == 1) {
```

```
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    // 创建新自由职业者客户端对象
    flclient_t *client = flclient_new ();

    // 连接到每个端点
    int argn;
    for (argn = 1; argn < argc; argn++)
        flclient_connect (client, argv [argn]);

    // 发送一批名字解析的“请求”，计时
    int requests = 10000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flclient_request (client, &request);
        if (!reply) {
            printf ("E: name service not available, aborting\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("Average round trip cost: %d usec\n",
        (int) (zclock_time () - start) / 10);

    flclient_destroy (&client);
    return 0;
}
```

flclient 类的实现如示例 4-82 所示。每个实例都有一个上下文，一个来跟我们的服务器交流的 DEALER 套接字，一个它连接到多少台服务器的计数器和一个请求序号。

示例4-82：自由职业者客户端，模型二（flclient2.c）：类实现

```
struct _flclient_t {
    zctx_t *ctx;          // 上下文包装器
    void *socket;         // 与服务器交流的 DEALER 套接字
    size_t servers;       // 已经连接的服务器数量
    uint sequence;        // 已发送的请求数量
};

// --------------------------------------------------------------------
// 构造函数
```

```
flclient_t *
flclient_new (void)
{
231     flclient_t
            *self;

    self = (flclient_t *) zmalloc (sizeof (flclient_t));
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_DEALER);
    return self;
}

//  --------------------------------------------------------------------
//  析构函数

void
flclient_destroy (flclient_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flclient_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  --------------------------------------------------------------------
//  连接到新服务器端点

void
flclient_connect (flclient_t *self, char *endpoint)
{
    assert (self);
    zsocket_connect (self->socket, endpoint);
    self->servers++;
}
```

示例 4-83 中的 request 方法执行繁重的工作。它发送一个请求到并联连接的所有服务器（要使这个能够工作，所有的连接必须都是成功的，并且要在这个时间内完成）。然后等待一个成功的回应，并返回给调用者。任何其他应答都只是被丢弃。

示例4-83：自由职业者客户端，模型二（flclient2.c）：请求方法

```
zmsg_t *
flclient_request (flclient_t *self, zmsg_t **request_p)
```

```
{
    assert (self);
    assert (*request_p);
    zmsg_t *request = *request_p;

    //  用序号和空封包作为请求前缀
    char sequence_text [10];
    sprintf (sequence_text, "%u", ++self->sequence);
    zmsg_pushstr (request, sequence_text);                232
    zmsg_pushstr (request, "");

    //  把请求发射到所有连接到的服务器
    int server;
    for (server = 0; server < self->servers; server++) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->socket);
    }
    //  等候从任何地方来的一个匹配的应答
    //  因为可以多次轮询，所以计算每个应答
    zmsg_t *reply = NULL;
    uint64_t endtime = zclock_time () + GLOBAL_TIMEOUT;
    while (zclock_time () < endtime) {
        zmq_pollitem_t items [] = { { self->socket, 0, ZMQ_POLLIN, 0 } };
        zmq_poll (items, 1, (endtime - zclock_time ()) * ZMQ_POLL_MSEC);
        if (items [0].revents & ZMQ_POLLIN) {
            //  应答是 [ 空 ][ 序号 ][OK]
            reply = zmsg_recv (self->socket);
            assert (zmsg_size (reply) == 3);
            free (zmsg_popstr (reply));
            char *sequence = zmsg_popstr (reply);
            int sequence_nbr = atoi (sequence);
            free (sequence);
            if (sequence_nbr == self->sequence)
                break;
        }
    }
    zmsg_destroy (request_p);
    return reply;
}
```

对于客户端实现，有下列注意事项：

- 客户端被构造为一个可爱的以类为基础的小 API，它隐藏了创建 ØMQ 上下文和套接字再跟服务器交流的肮脏的工作。也就是说，假设用猎枪射击腹部可以被称为“交流”。

- 如果客户端在几秒之内不能找到任何响应的服务器，它将放弃追逐。
- 客户端必须建立一个有效的 REP 封包（即把一个空消息帧添加到消息的前面）。

客户端执行 10000 个解析请求（假的，因为我们的服务器基本上什么也不做），并测量平均成本。在我的测试电脑中，与一台服务器交流，大约需要 60 微秒。与三台服务器交流，大约需要 80 微秒。

猎枪法的优点和缺点分别如下所述。

233
- 优点：简单，容易制造，容易理解。
- 优点：只要至少有一台服务器在运行，它就执行故障转移工作，并且执行速度很快。
- 缺点：产生冗余的网络流量。
- 缺点：不能对我们的服务器排定优先级（即先主服务器，然后是备用服务器）。
- 缺点：服务器在同一时间最多可以处理一个请求。

模式三：复杂和讨厌的

猎枪法似乎是好得不真实的。让我们更科学化，并遍历所有的替代方法。我们要探索复杂的 / 讨厌的选项，即使只是为了最终认识到我们要首选粗暴的做法。啊，这就是生活。

我们可以通过切换到一个 ROUTER 套接字来解决客户端的主要问题。它允许我们将请求发送到特定的服务器，避免发送到明知道是"死的"服务器，这一般被视为与我们想要的一样聪明。我们还可以通过切换到 ROUTER 套接字解决服务器（单线程）的主要问题。

但在两个匿名套接字（未设置标识）之间做 ROUTER 到 ROUTER 的连接是不可能的。双方都只有当它们收到第一个消息时才产生另一个节点的身份，从而任何一方直到它第一次收到消息前都不能跟另一方交流。解决这一难题的唯一出路是欺骗，并在一个方向上使用硬编码的身份。在客户端 / 服务器的情况下，正确的欺骗方法是让客户端"知道"服务器的身份。对于复杂和讨厌的选项，用其他方式来完成它将是疯狂的，因为能够独立产生任何数量的客户端。疯狂、复杂和讨厌，对于一个种族灭绝的独裁者来说，它们是伟大的属性，但对于软件，它们是可怕的属性。

我们将使用连接端点作为身份，而不是创造另一种概念来管理。对于猎枪模型，这是双方都同意的一个唯一的字符串，而不需要它们掌握比其已有的更多的先验知识。这是用来连接两个 ROUTER 套接字的聪明和有效的方式。

请回顾 ØMQ 身份是如何工作的。服务器 ROUTER 套接字先设置了一个身份，然后将它绑定到其套接字。当客户端连接时，在任何一方发送一个真正的消息前，它们都要先做一点点握手来交换身份。客户端 ROUTER 套接字，还没有设置一个身份，它将发送一

个空的身份给服务器。服务器生成一个随机 UUID 来指定客户端，供它自己使用。服务器发送其身份（我们已经同意它将是一个端点字符串）到客户端。

这意味着，一旦建立连接，我们的客户端就可以将消息路由到服务器（即发送到其 ROUTER 套接字，指定服务器端点作为身份）。这不是在执行一个 zmq_connect() 后立即进行的，而是在其后某个随机的时间进行的。这里存在一个问题：我们不知道什么时候服务器实际上可用并完成其连接握手。如果服务器处于联机状态，它可能是几毫秒后。如果服务器宕机了，而系统管理员又出去吃午饭了，它可能是从现在起的一个小时后。 234

这里有一个小的矛盾。我们需要知道服务器什么时候变得已连接并可用于工作。在自由职业者模式中，不像我们在本章前面看到的以代理为基础的模式，服务器直到发言前都是沉默的。因此，我们不能与一台服务器交流，除非它告诉我们它已联机，而它又不能告诉我们这个，除非我们询问它。

我的解决方案是把猎枪法和模型二进行少量组合，这意味着将对我们可以瞄准的任何东西发射（无害的）猎枪弹，并且如果有什么东西动了，我们就知道它是活的。我们不会发射真正的请求，而只是发射一种乒乓信号用来检测。



这再次将我们带到协议的境界：你会在 *http://rfc.zeromq.org/spec:10* 发现一个简短的规范，它定义了一个自由职业者客户端和服务器如何交换乒乓命令和请求 - 应答命令。

作为服务器来实现是短暂而甜蜜的。示例 4-84 展示了回应服务器的模型三的代码，现在称为自由职业者协议（FLP）。

示例4-84：自由职业者服务器，模型三（flserver3.c）

```
//
//  自由职业者服务器 - 模型三
//  使用一个 ROUTER/ROUTER 套接字，但只有一个线程
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    zctx_t *ctx = zctx_new ();

    //  用预知的身份准备服务器套接字
    char *bind_endpoint = "tcp://*:5555";
```

```
    char *connect_endpoint = "tcp://localhost:5555";
    void *server = zsocket_new (ctx, ZMQ_ROUTER);
    zmq_setsockopt (server,
        ZMQ_IDENTITY, connect_endpoint, strlen (connect_endpoint));
    zsocket_bind (server, bind_endpoint);
    printf ("I: service is ready at %s\n", bind_endpoint);

    while (!zctx_interrupted) {
        zmsg_t *request = zmsg_recv (server);
        if (verbose && request)
235         zmsg_dump (request);
        if (!request)
            break;          //  中断

        //  第 0 帧：客户端身份
        //  第 1 帧：PING 或客户端控制帧
        //  第 2 帧：请求的正文
        zframe_t *identity = zmsg_pop (request);
        zframe_t *control = zmsg_pop (request);
        zmsg_t *reply = zmsg_new ();
        if (zframe_streq (control, "PING"))
            zmsg_addstr (reply, "PONG");
        else {
            zmsg_add (reply, control);
            zmsg_addstr (reply, "OK");
        }
        zmsg_destroy (&request);
        zmsg_push (reply, identity);
        if (verbose && reply)
            zmsg_dump (reply);
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

但是，自由职业者客户端已经变得很庞大。为了清楚起见，将它分解成一个示例应用程序和一个执行繁重工作的类。顶级的应用程序如示例 4-85 所示。

示例4-85：自由职业者客户端，模型三（flclient3.c）

```
//
//  自由职业者客户端 - 模型三
```

```
//  使用flcliapi类来封包自由职业者模式
//
//  构建这个源代码，但不创建一个库
#include "flcliapi.c"

int main (void)
{
    //  创建新自由职业者客户端对象
    flcliapi_t *client = flcliapi_new ();

    //  连接到每个端点
    flcliapi_connect (client, "tcp://localhost:5555");
    flcliapi_connect (client, "tcp://localhost:5556");
    flcliapi_connect (client, "tcp://localhost:5557");

    //  发送一批名字解析的“请求”，计时
    int requests = 1000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flcliapi_request (client, &request);
        if (!reply) {
            printf ("E: name service not available, aborting\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("Average round trip cost: %d usec\n",
        (int) (zclock_time () - start) / 10);

    flcliapi_destroy (&client);
    return 0;
}
```

236

示例 4-86 给出了客户端 API 类，它几乎与管家代理同样复杂和庞大。

示例4-86：自由职业者客户端API（flcliapi.c）

```
/*  =====================================================================
 *  flcliapi - 自由职业者模式代理类
 *  实现了在 http://rfc.zeromq.org/spec:10 的自由职业者协议
 *  ===================================================================== */

#include "flcliapi.h"

//  如果没有服务器在此时间内应答，就放弃请求
```

```
#define GLOBAL_TIMEOUT  3000    //  毫秒
//  我们认为在运行的服务器的 PING 时间间隔
#define PING_INTERVAL   2000    //  毫秒
//  如果静默了这么长时间，就认为服务器宕机
#define SERVER_TTL      6000    //  毫秒
```

这个 API 在两个一半的部分中工作，这是那些需要在后台运行的 API 采用的一个共同的模式。其中一半是我们的应用程序创建并处理的一个前端对象，另一半是在后台线程中运行的一个后端“代理”。前端通过一个 inproc 管道套接字与后端交流。该 API 结构如示例 4-87 所示。

示例4-87：自由职业者客户端API（flcliapi.c）：API结构

```
//  ---------------------------------------------------------------------
//  前端类的结构

struct _flcliapi_t {
    zctx_t *ctx;        //  上下文包装器
    void *pipe;         //  通向 flcliapi 代理的管道
237 };

//  这是处理实际 flcliapi 类的线程
static void flcliapi_agent (void *args, zctx_t *ctx, void *pipe);

//  ---------------------------------------------------------------------
//  构造函数

flcliapi_t *
flcliapi_new (void)
{
    flcliapi_t
        *self;

    self = (flcliapi_t *) zmalloc (sizeof (flcliapi_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, flcliapi_agent, NULL);
    return self;
}

//  ---------------------------------------------------------------------
//  析构函数

void
flcliapi_destroy (flcliapi_t **self_p)
{
```

```
    assert (self_p);
    if (*self_p) {
        flcliapi_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}
```

为了实现 connect 方法（参见示例 4-88），前端对象发送一个多部分消息给后端代理。第一部分是一个字符串“CONNECT”，而第二部分是端点。它为到达的连接等待 100 毫秒，虽然这不漂亮，但让我们可以防止在启动时将所有的请求发送到一台服务器上。

示例4-88：自由职业者客户端API（flcliapi.c）：连接方法

```
void
flcliapi_connect (flcliapi_t *self, char *endpoint)
{
    assert (self);
    assert (endpoint);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, endpoint);
    zmsg_send (&msg, self->pipe);
    zclock_sleep (100);      //  允许连接到达
}
```

238

为了实现 request 方法，前端对象将消息发送到后端，指定一个命令“REQUEST”和请求消息（参见示例 4-89）。

示例4-89：自由职业者客户端API（flcliapi.c）：请求方法

```
zmsg_t *
flcliapi_request (flcliapi_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);

    zmsg_pushstr (*request_p, "REQUEST");
    zmsg_send (request_p, self->pipe);
    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *status = zmsg_popstr (reply);
        if (streq (status, "FAILED"))
            zmsg_destroy (&reply);
        free (status);
```

```
    }
    return reply;
}
```

现在，让我们来看看后台代理。它作为一个附加的线程运行，通过一个管道套接字与它的父线程交流。这是一项相当复杂的工作，所以我们将它分解成部件。首先，代理用我们熟悉的类的方法管理一组服务器（参见示例 4-90）。

示例4-90：自由职业者客户端API（flcliapi.c）：后端代理

```
//  --------------------------------------------------------------------
//  用于我们要交流的一台服务器的简单类

typedef struct {
    char *endpoint;             //  服务器身份 / 端点
    uint alive;                 //  如果知道正在运行，值为 1
    int64_t ping_at;            //  在这个时间执行下一个 ping
    int64_t expires;            //  在这个时间过期
} server_t;

server_t *
server_new (char *endpoint)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));
    self->endpoint = strdup (endpoint);
    self->alive = 0;
    self->ping_at = zclock_time () + PING_INTERVAL;
    self->expires = zclock_time () + SERVER_TTL;
    return self;
}
```
239
```
void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->endpoint);
        free (self);
        *self_p = NULL;
    }
}

int
server_ping (const char *key, void *server, void *socket)
{
```

```
    server_t *self = (server_t *) server;
    if (zclock_time () >= self->ping_at) {
        zmsg_t *ping = zmsg_new ();
        zmsg_addstr (ping, self->endpoint);
        zmsg_addstr (ping, "PING");
        zmsg_send (&ping, socket);
        self->ping_at = zclock_time () + PING_INTERVAL;
    }
    return 0;
}

int
server_tickless (const char *key, void *server, void *arg)
{
    server_t *self = (server_t *) server;
    uint64_t *tickless = (uint64_t *) arg;
    if (*tickless > self->ping_at)
        *tickless = self->ping_at;
    return 0;
}
```

我们将代理构建为一个能够处理从它的各种套接字进来的消息的类，如示例 4-91 所示。

示例4-91：自由职业者客户端API（flcliapi.c）：后端代理类

```
//  ---------------------------------------------------------------------
//  用于一个后台代理的简单类

typedef struct {
    zctx_t *ctx;                //  自己的上下文
    void *pipe;                 //  回去与应用程序交流的套接字
    void *router;               //  与服务器交流的套接字
    zhash_t *servers;           //  已经连接到的服务器
    zlist_t *actives;           //  已知在运行的服务器
    uint sequence;              //  发送的请求数量
    zmsg_t *request;            //  当前请求（如果有）
    zmsg_t *reply;              //  当前应答（如果有）
    int64_t expires;            //  请求 / 应答的超时时间
} agent_t;

agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->router = zsocket_new (self->ctx, ZMQ_ROUTER);
```

240

```
    self->servers = zhash_new ();
    self->actives = zlist_new ();
    return self;
}

void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        zhash_destroy (&self->servers);
        zlist_destroy (&self->actives);
        zmsg_destroy (&self->request);
        zmsg_destroy (&self->reply);
        free (self);
        *self_p = NULL;
    }
}
```

control_message方法，如示例4-92所示，处理来自前端类的一个消息（这将是"CONNECT"或"REQUEST"）。

示例4-92：自由职业者客户端API（flcliapi.c）：控制消息

```
//  当我们将服务器从代理的"servers"散列表删除时回调

static void
s_server_free (void *argument)
{
    server_t *server = (server_t *) argument;
    server_destroy (&server);
}

void
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);

241 if (streq (command, "CONNECT")) {
        char *endpoint = zmsg_popstr (msg);
        printf ("I: connecting to %s...\n", endpoint);
        int rc = zmq_connect (self->router, endpoint);
        assert (rc == 0);
        server_t *server = server_new (endpoint);
```

```
        zhash_insert (self->servers, endpoint, server);
        zhash_freefn (self->servers, endpoint, s_server_free);
        zlist_append (self->actives, server);
        server->ping_at = zclock_time () + PING_INTERVAL;
        server->expires = zclock_time () + SERVER_TTL;
        free (endpoint);
    }
    else
    if (streq (command, "REQUEST")) {
        assert (!self->request);    //  严格的请求 - 应答循环
        //  用序号和空封包作为请求前缀
        char sequence_text [10];
        sprintf (sequence_text, "%u", ++self->sequence);
        zmsg_pushstr (msg, sequence_text);
        //  取得请求消息的所有权
        self->request = msg;
        msg = NULL;
        //  请求在全局超时时间后过期
        self->expires = zclock_time () + GLOBAL_TIMEOUT;
    }
    free (command);
    zmsg_destroy (&msg);
}
```

router_message 方法，如示例 4-93 所示，处理来自连接的服务器的一个消息。

示例4-93：自由职业者客户端API（flcliapi.c）：路由消息

```
void
agent_router_message (agent_t *self)
{
    zmsg_t *reply = zmsg_recv (self->router);

    //  第 0 帧是应答的服务器
    char *endpoint = zmsg_popstr (reply);
    server_t *server =
        (server_t *) zhash_lookup (self->servers, endpoint);
    assert (server);
    free (endpoint);
    if (!server->alive) {
        zlist_append (self->actives, server);
        server->alive = 1;
    }
    server->ping_at = zclock_time () + PING_INTERVAL;
    server->expires = zclock_time () + SERVER_TTL;

    //  第 1 帧可以是应答序号
```
242

```
    char *sequence = zmsg_popstr (reply);
    if (atoi (sequence) == self->sequence) {
        zmsg_pushstr (reply, "OK");
        zmsg_send (&reply, self->pipe);
        zmsg_destroy (&self->request);
    }
    else
        zmsg_destroy (&reply);
}
```

最后，示例 4-94 显示了代理任务本身，它轮询其两个套接字并处理进来的消息。

示例4-94：自由职业者客户端API（flcliapi.c）：后端代理实现

```
static void
flcliapi_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    zmq_pollitem_t items [] = {
        { self->pipe, 0, ZMQ_POLLIN, 0 },
        { self->router, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        //  用无滴答计时器计算时间，可长达 1 小时
        uint64_t tickless = zclock_time () + 1000 * 3600;
        if (self->request
        &&  tickless > self->expires)
            tickless = self->expires;
        zhash_foreach (self->servers, server_tickless, &tickless);

        int rc = zmq_poll (items, 2,
            (tickless - zclock_time ()) * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  上下文已被关闭

        if (items [0].revents & ZMQ_POLLIN)
            agent_control_message (self);

        if (items [1].revents & ZMQ_POLLIN)
            agent_router_message (self);

        //  如果我们正在处理请求，就将它分发到下一台服务器
        if (self->request) {
            if (zclock_time () >= self->expires) {
                //  请求过期，就清除它
                zstr_send (self->pipe, "FAILED");
```

```
                zmsg_destroy (&self->request);
            }
            else {
                //  找到要与之交流的服务器，删除任何过期的服务器
                while (zlist_size (self->actives)) {
                    server_t *server =
                        (server_t *) zlist_first (self->actives);
                    if (zclock_time () >= server->expires) {
                        zlist_pop (self->actives);
                        server->alive = 0;
                    }
                    else {
                        zmsg_t *request = zmsg_dup (self->request);
                        zmsg_pushstr (request, server->endpoint);
                        zmsg_send (&request, self->router);
                        break;
                    }
                }
            }
        }
        //  断开连接并删除任何过期的服务器
        //  如有必要，给空闲的服务器发送检测信号
        zhash_foreach (self->servers, server_ping, self->router);
    }
    agent_destroy (&self);
}
```

这个 API 的实现是相当复杂的，并使用了一些我们以前没有见过的技术。

多线程的 API

客户端 API 由两部分组成：一部分是运行在应用程序线程的同步 flcliapi 类，另一部分是作为一个后台线程运行的异步 agent 类。记住 ØMQ 如何可以轻松地创建多线程应用程序。该 flcliapi 和 agent 类通过一个 inproc 套接字用消息相互交流。所有 ØMQ 方面（如创建和销毁上下文）都隐藏在这个 API 中。实际上，代理的作用就像一个小型的代理，与在后台运行的服务器交流，这样当我们发出一个请求时，它就可以尽最大努力使之到达它认为是可用的服务器。

无滴答轮询计时器

在前面的轮询循环中，我们总是用一个固定的刻度间隔，如 1 秒作为轮询延迟时间。这虽然很简单，但对于对功耗敏感的客户端（如笔记本电脑和移动电话）并不出色，因为在那里唤醒 CPU 要消耗电量。为了好玩，并帮助拯救地球，代理使用一个“无滴答计时器”，它基于我们期待的下一次超时时间计算轮询延迟时间。一个适当的实

现将保持一个超时时间的有序列表，我们只是检查所有超时时间并计算直到下一次轮询的延迟时间。

244
结论

在本章中，我们看到了各种可靠的请求 - 应答机制，每种都具有一定的成本和效益。示例代码很大程度是为实际使用准备的，虽然它不是最优化的。在所有不同的模式中，两种突出的可投入生产使用的模式是用于以代理为基础的可靠性的管家模式，以及用于无代理可靠性的自由职业者模式。

第 5 章 高级发布-订阅模式

在第 3 章和第 4 章中，我们看到了 ØMQ 的请求 - 应答模式的高级使用。如果你成功地消化了这一切，那么向你表示祝贺。在本章中，我们将专注于发布 - 订阅模式，并且用更高级别的模式扩展 ØMQ 的核心发布 - 订阅模式以改善性能、可靠性、状态分布和监控方面。

我们将讨论：

- 何时使用发布 - 订阅模式
- 如何处理过于慢速的订阅者（*自杀蜗牛模式*）
- 如何设计高速订阅者（*黑盒模式*）
- 如何监控一个发布 - 订阅网络（*特浓咖啡模式*）
- 如何建立一个共享的键 - 值存储（*克隆模式*）
- 如何用反应器来简化复杂的服务器
- 如何使用双星模式把故障转移添加到一台服务器

发布 - 订阅模式的优点和缺点

ØMQ 的低级别的模式各自都有不同的特点。发布 - 订阅模式解决了旧的消息传递问题，这就是*多播*（*multicast*）或*群发消息*（*group messaging*）。这个模式把描述 ØMQ 特点的细致的简单性和粗暴冷漠进行了独特的组合。因此，有必要了解发布 - 订阅模式做出了哪些权衡、这些权衡如何造福于我们，以及如果需要的话我们应该如何解决它们。

首先，PUB 将每条消息发送给“全部”接收者，而 PUSH 和 DEALER 只将消息转给“其中之一”。不能简单地用 PUB 替换 PUSH 并希望事情会正常工作，反之亦然。这句话值得反复强调，因为人们似乎常常以为这样做能正常工作。

246 更深刻的，发布 - 订阅模式是专门针对可扩展性的。这意味着将大量的数据快速地发送给许多接收者。如果你需要每秒将数百万的消息发送到几千个节点，而不是将少量消息在一秒内发送到屈指可数的接收者，你就会非常感谢发布 - 订阅模式。

为了获得可扩展性，发布 - 订阅模式使用与推送 - 提取模式相同的技巧，这就是去掉喋喋不休的回答。这意味着，接收者不回应发送者。但也有一些例外，例如，SUB 套接字会发送订阅给 PUB 套接字，但这种发送是匿名的并且很少有。

去掉喋喋不休的回答对真正的可扩展性是必不可少的。发布 - 订阅这个模式怎么才能清晰地映射到实际通用多播（PGM）协议呢？这是由网络交换机处理的。换句话说，订阅者根本不连接到发布者，它们连接到交换机上的多播组，而发布者将其消息发送给该多播组。

当我们去掉喋喋不休的回答时，整体的信息流变得简单多了，这就可以让我们做出更简单的 API，更简单的协议，并且一般来说能把消息送达更多的人。但是，我们也失去了协调发送者和接收者的任何可能性。这句话的意思是：

- 发布者不能判断出订阅者何时成功连接，无论是初始连接还是网络故障后的重新连接。
- 订阅者不能告诉发布者任何东西来允许发布者控制它们发送消息的速度。发布者只能有一个设置，这就是*全速*，而订阅者必须要么跟上这个速度，要么丢失消息。
- 发布者不能判断出订阅者何时由于程序崩溃，网络中断等原因而消失。

不利的一面是，如果我们想要执行可靠的多播，确实需要所有这些功能。当订阅者正在连接时，当网络发生故障时，或者只是如果订阅者或网络无法跟上发布者的速度时，ØMQ 的发布 - 订阅模式都将任意地丢失消息。

有利的一面是，有许多*接近*可靠的多播还是很好的用例。当我们需要这个喋喋不休的回答时，要么我们可以切换到使用 ROUTER-DEALER（对于大部分正常数量的用例，我倾向于这么做），要么我们可以添加一个单独的通道进行同步（会在本章后面看到这样的例子）。

发布 - 订阅模式就像是一个无线电广播：你错过了加入前的一切东西，然后你获得多少信息取决于你的接收质量。令人惊讶的是，这种模式是有用并且普遍的，因为它完美地映射了现实世界的信息分布。请考虑 Facebook 和 Twitter，英国广播公司世界服务，以及体育比赛结果。

247 正如我们对请求 - 应答模式所做的，让我们用可能出现什么问题来定义*可靠性*。下面是发布 - 订阅模式的经典失败案例：

- 订阅者加入晚了，所以错过了服务器已经发送的消息。
- 订阅者获取消息的速度可能太慢，所以队列堆积起来，然后溢出。
- 订阅者可能脱落而丢失当它们离开时的消息。
- 订阅者可能崩溃并重新启动，而丢失它们已经收到的任何数据。
- 网络可能变得超载并删除数据（具体而言，对于 PGM）。
- 网络可能变得太慢，所以发布者一方队列溢出而发布者崩溃。

还有很多情况可能出错，但这些都是我们在现实系统中看到的典型故障。从 v3.x 版本起，ØMQ 强制对其内部缓冲区设置默认的限值（即所谓的高水位标记或 HWM），所以发布者崩溃是罕见的，除非你故意把 HWM 设置为无限。

所有这些失败的案例都有解决办法，但并不总是简单的。可靠性要求的复杂性，我们大多数人在大部分时间里并不需要它，这就是为什么 ØMQ 不会尝试预置这个功能的原因。

发布 - 订阅跟踪（特浓咖啡模式）

让我们通过观察一个跟踪发布 - 订阅网络的方法来开始这一章。在第 2 章中，我们看到过用这些方法做传输桥接的一个简单的代理。`zmq_proxy()` 方法有三个参数：桥接在一起的前端（*frontend*）套接字和后端（*backend*）套接字，还有一个把所有消息都发到其上的捕获（*capture*）套接字。

该代码看似简单，如示例 5-1 所示。

示例5-1：特浓咖啡模式（espresso.c）

```
//
//  特浓咖啡模式
//  这显示如何用发布 - 订阅代理来捕获数据
//
#include "czmq.h"

//  订阅者请求以 A 或 B 开始的消息，
//  然后读取进来的消息并对其计数。

static void
subscriber_thread (void *args, zctx_t *ctx, void *pipe)
{
    //  订阅 "A" 和 "B"
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:6001");
    zsockopt_set_subscribe (subscriber, "A");
    zsockopt_set_subscribe (subscriber, "B");
```

248

```
    int count = 0;
    while (count < 5) {
        char *string = zstr_recv (subscriber);
        if (!string)
            break;              //  中断
        free (string);
        count++;
    }
    zsocket_destroy (ctx, subscriber);
}
```

发布者随机发送以 A ~ J 开始的消息，如示例 5-2 所示。

示例5-2：特浓咖啡模式（espresso.c）：发布者线程

```
static void
publisher_thread (void *args, zctx_t *ctx, void *pipe)
{
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:6000");

    while (!zctx_interrupted) {
        char string [10];
        sprintf (string, "%c-%05d", randof (10) + 'A', randof (100000));
        if (zstr_send (publisher, string) == -1)
            break;              //  中断
        zclock_sleep (100);     //  等待十分之一秒
    }
}
```

如示例 5-3 所示的监听器负责接收流经代理的所有消息。在 CZMQ 中，管道是一对 ZMQ_PAIR 套接字，它连接附加的子线程。在其他语言中，你获得的好处可能会有所不同。

示例5-3：特浓咖啡模式（espresso.c）：监听器线程

```
static void
listener_thread (void *args, zctx_t *ctx, void *pipe)
{
    //  输出到达管道的所有东西
    while (true) {
        zframe_t *frame = zframe_recv (pipe);
        if (!frame)
            break;              //  中断
        zframe_print (frame, NULL);
        zframe_destroy (&frame);
    }
}
```

主任务（参见示例 5-4）启动订阅者和发布者，然后把它自己设为监听代理。监听器作为子线程运行。 249

示例5-4：特浓咖啡模式（espresso.c）：主线程

```
int main (void)
{
    //  启动子线程
    zctx_t *ctx = zctx_new ();
    zthread_fork (ctx, publisher_thread, NULL);
    zthread_fork (ctx, subscriber_thread, NULL);

    void *subscriber = zsocket_new (ctx, ZMQ_XSUB);
    zsocket_connect (subscriber, "tcp://localhost:6000");
    void *publisher = zsocket_new (ctx, ZMQ_XPUB);
    zsocket_bind (publisher, "tcp://*:6001");
    void *listener = zthread_fork (ctx, listener_thread, NULL);
    zmq_proxy (subscriber, publisher, listener);

    puts (" interrupted");
    //  通知附加线程退出
    zctx_destroy (&ctx);
    return 0;
}
```

特浓咖啡的工作原理是创建一个监听线程读取一个 PAIR 套接字，并输出得到的任何东西。管道的一端是一个 PAIR 套接字，另一端（另一个 PAIR）是我们传递给 `zmq_proxy()` 的套接字。在实践中，你会筛选有趣的消息，得到你所要跟踪的精华（这就是该模式的名称的由来）。

订阅者线程订阅“A”和“B”，接收 5 条消息，然后销毁它的套接字。当你运行一个实例时，监听器输出 2 条订阅消息，5 条数据消息，2 条退订消息，然后沉默：

```
[002] 0141
[002] 0142
[007] B-91164
[007] B-12979
[007] A-52599
[007] A-06417
[007] A-45770
[002] 0041
[002] 0042
```

这整齐地表明，当发布者套接字没有订阅者时，它如何停止发送数据。发布者线程仍在发送消息。套接字只是默默地删除它们。

250

最后一个值缓存

如果你已经使用了商业发布 - 订阅系统，可能会使用一些在快速和欢快的 ØMQ 发布 - 订阅模型中缺少的功能。其中之一就是“*最后一个值缓存（LVC）*”。这解决了一个新订阅者加入网络时，它如何追上的问题。该理论认为，当一个新的订阅者加入，并订阅了一些特定的主题时，发布者得到通知。然后，发布者可以重新广播这些主题的最后一条消息。

我已经解释了为什么当有新订阅者时发布者没有得到通知：在大型发布 - 订阅系统中的数据量使得它几乎是不可能的。为了使真正大规模的发布 - 订阅网络正常工作，你需要类似于 PGM 的协议，它利用一个高档以太网交换机的能力将数据多播到成千上万的订阅者。试图从发布者到成千上万的订阅者中的每个都执行一个 TCP 单播是不能扩展的。你会得到奇怪的尖峰，分配不公平（一些订阅者在其他订阅者之前收到消息），网络拥塞，和一般的不快。

PGM 是一个单向协议：发布者将消息发送到交换机上的一个多播地址，然后交换机为所有感兴趣的订阅者转播。发布者永远不会看到订阅者何时加入或离开：这一切都发生在交换机上，我们真的不希望在它上面开始重新编程。

然而，在只有几十个订阅者和数量有限的主题的低容量的网络上，我们可以使用 TCP，然后 XSUB 和 XPUB 套接字确实互相交流，就像我们刚刚在特浓咖啡模式中看到的。

我们可以使用 ØMQ 来制作最后的值缓存吗？答案是肯定的，如果在发布者和订阅者之间制作一个代理，也就是一个模拟的 PGM 交换机，但它是一个我们可以自己编程的交换机。

我们将通过制作一个突出最坏情况的发布者和订阅者开始。该发布者是病态的，它通过立即将消息发送到 1000 个主题中的每个来开始，然后它每秒发送一个更新到随机的主题。订阅者连接并订阅一个主题。如果没有 LVC，订阅者将不得不等待平均 500 秒才能获得任何数据。要添加一些戏剧效果，让我们假设有一个逃犯叫格里，他威胁说，如果我们不能修复那个 8.3 分钟（译者注：即前文提到的 500 秒）的延迟，他就拧下罗杰的玩具兔子的头。

示例 5-5 给出了发布者的源代码。请注意，它有一个命令行选项来连接某个地址，否则就绑定到端点。以后我们将使用这个选项来连接最后一个值缓存。

示例5-5：病态发布者（pathopub.c）

```
//
// 病态发布者
// 发出1000个主题，然后每秒随机更新一个
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    zctx_t *context = zctx_new ();
    void *publisher = zsocket_new (context, ZMQ_PUB);
    if (argc == 2)
        zsocket_connect (publisher, argv [1]);
    else
        zsocket_bind (publisher, "tcp://*:5556");

    // 确保订阅者连接有时间来完成
    sleep (1);

    // 发出全部1000个主题消息
    int topic_nbr;
    for (topic_nbr = 0; topic_nbr < 1000; topic_nbr++) {
        zstr_sendfm (publisher, "%03d", topic_nbr, ZMQ_SNDMORE);
        zstr_send (publisher, "Save Roger");
    }
    // 每秒发送一个随机更新
    srandom ((unsigned) time (NULL));
    while (!zctx_interrupted) {
        sleep (1);
        zstr_sendfm (publisher, "%03d", randof (1000), ZMQ_SNDMORE);
        zstr_send (publisher, "Off with his head!");
    }
    zctx_destroy (&context);
    return 0;
}
```

251

订阅者的代码如示例5-6所示。

示例5-6：病态订阅者（pathosub.c）

```
//
// 病态订阅者
// 订阅了一个随机的话题，并输出接收到的消息
//
#include "czmq.h"
```

```
int main (int argc, char *argv [])
{
    zctx_t *context = zctx_new ();
    void *subscriber = zsocket_new (context, ZMQ_SUB);
    if (argc == 2)
        zsocket_connect (subscriber, argv [1]);
    else
        zsocket_connect (subscriber, "tcp://localhost:5556");

    srandom ((unsigned) time (NULL));
    char subscription [5];
252 sprintf (subscription, "%03d", randof (1000));
    zsocket_set_subscribe (subscriber, subscription);

    while (true) {
        char *topic = zstr_recv (subscriber);
        if (!topic)
            break;
        char *data = zstr_recv (subscriber);
        assert (streq (topic, subscription));
        puts (data);
        free (topic);
        free (data);
    }
    zctx_destroy (&context);
    return 0;
}
```

尝试建立并运行这些:首先是订阅者,然后是发布者。你会如你所愿地看到,订阅者报告,他得到了“拯救罗杰”:

```
./pathosub &
./pathopub
```

当你运行第二个订阅者时,你明白了罗杰的困境是:在它报告得到任何数据前,你必须等待相当长的时间。最后一个值缓存呈现在示例 5-7 到示例 5-9 中。正如我答应的,这是一个绑定到两个套接字,然后处理两者上面的消息的代理。

示例5-7:最后一个值缓存代理(lvcache.c)

```
//
//  最后一个值缓存
//  使用 XPUB 订阅消息重新发送数据
//
#include "czmq.h"
```

```
int main (void)
{
    zctx_t *context = zctx_new ();
    void *frontend = zsocket_new (context, ZMQ_SUB);
    zsocket_bind (frontend, "tcp://*:5557");
    void *backend = zsocket_new (context, ZMQ_XPUB);
    zsocket_bind (backend, "tcp://*:5558");

    //  从发布者订阅每一个话题
    zsocket_set_subscribe (frontend, "");

    //  在缓存中存储每个主题的最后一个实例
    zhash_t *cache = zhash_new ();
```

我们将主题的更新从前端路由至后端，并且通过发送我们缓存的任何东西（如果有）来处理订阅，如示例 5-8 所示。

示例5-8：最后一个值缓存代理（lvcache.c）：主轮询循环 253

```
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { backend,  0, ZMQ_POLLIN, 0 }
        };
        if (zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC) == -1)
            break;              //  中断

        //  我们缓存任何新主题数据然后转发
        if (items [0].revents & ZMQ_POLLIN) {
            char *topic = zstr_recv (frontend);
            char *current = zstr_recv (frontend);
            if (!topic)
                break;
            char *previous = zhash_lookup (cache, topic);
            if (previous) {
                zhash_delete (cache, topic);
                free (previous);
            }
            zhash_insert (cache, topic, current);
            zstr_sendm (backend, topic);
            zstr_send (backend, current);
            free (topic);
        }
```

当得到一个新的订阅时，我们把数据从缓存中取出，如示例 5-9 所示。

示例5-9：最后一个值缓存代理（lvcache.c）：处理订阅

```
            zframe_t *frame = zframe_recv (backend);
            if (!frame)
                break;
            //  事件是一个字节，0=unsub 或 1=sub，跟在主题后面
            byte *event = zframe_data (frame);
            if (event [0] == 1) {
                char *topic = zmalloc (zframe_size (frame));
                memcpy (topic, event + 1, zframe_size (frame) - 1);
                printf ("Sending cached topic %s\n", topic);
                char *previous = zhash_lookup (cache, topic);
                if (previous) {
                    zstr_sendm (backend, topic);
                    zstr_send (backend, previous);
                }
                free (topic);
            }
            zframe_destroy (&frame);
        }
    }
    zctx_destroy (&context);
    zhash_destroy (&cache);
254 return 0;
}
```

现在，先运行代理，然后运行发布者：

```
./lvcache &
./pathopub tcp://localhost:5557
```

然后，尝试运行尽可能多的订阅者实例，每次都连接到代理的 5558 端口：

```
./pathosub tcp://localhost:5558
```

每个订阅者都高兴地报告“拯救罗杰”，而逃犯格里偷偷溜回到自己的座位吃饭，喝一杯不错的热牛奶，这都与他的初衷吻合。

注意：默认情况下，XPUB 套接字不报告重复的订阅，当你天真地连接一个 XPUB 到 XSUB 时，这是你想要的。我们的例子中通过使用随机的主题悄悄地解决了这个问题，所以它不能工作的机会是百万分之一。在实际的 LVC 代理中，你会希望使用在第 6 章 ØMQ 社区中作为一个练习实现的 ZMQ_XPUB_VERBOSE 选项。

慢速订阅者检测（自杀蜗牛模式）

在现实中使用发布 - 订阅模式的时候，你将面临的一个常见问题是慢速订阅者。在一个理想的世界里，我们将数据从发布者全速发送给订阅者。在现实中，订阅者应用程序通常用解释型语言编写，要么做了很多工作，要么只是编写得不好，以至于它们无法跟上发布者的速度。

如何处理一个慢速订阅者呢？理想的解决办法是让订阅者的速度更快，但是这可能需要相当大的工作量和时间才能做到。处理一个慢速订阅者有下面一些经典策略。

- *在发布方排队消息*。这是当我几小时不读我的电子邮件后，Gmail 所做的。但在大批量的消息传递中，逆向推送队列具有惊心动魄但无利可图的结果，它使发布者耗尽内存，然后崩溃，尤其是如果有大量的订阅者，而且出于性能原因是不可能刷新到磁盘的时候。
- *在订阅方排队消息*。这个策略好多了，这就是 ØMQ 在默认情况下所做的，如果网络能够跟上消息流速的话。如果有任何人耗尽内存并崩溃，这将是订阅者，而不是发布者，这是公平的。对于某个订阅者有一段时间跟不上流速，但当流减慢时它能赶上“有峰”流，这是完美的。然而，它没有解决订阅者只是普遍的速度太慢的问题。
- *一段时间后停止对新消息排队*。这是当我的邮箱溢出其珍贵的千兆字节的空间时，255
Gmail 所做的。新的消息只是被拒绝或丢弃。从发布者的角度来看，这是一个伟大的策略，这就是发布者设置一个 HWM 时，ØMQ 所做的。但是，它仍然无助于我们解决慢速订阅者，现在我们只是在消息流中得到间隙。
- *用断开连接惩治慢速订阅者*。这是当我两个星期没登录时，Hotmail（还记得它吗？）所做的，这就是为什么我在我的第 15 个 Hotmail 账户上的原因（译者注：前 14 个账户都被 Hotmail 删除了），它可能有另一个更好的办法来打击我。它强制订阅者坐起来，并注意听讲，这是一个不错的、粗暴的策略，它非常适合这种情况。然而，ØMQ 不这样做，而且也没有办法在顶层这么做，因为对于发布者应用程序，订阅者是不可见的。

因为这些经典策略没有适合的，所以我们需要发挥创意。让我们说服订阅者来杀死自己，而不是断开发布者的连接。这就是自杀蜗牛模式。当订阅者检测到它自己的运行速度过慢（其中“过慢”大概是一个配置选项，它的真正含义是：如此缓慢，以至于如果你有机会在这里，请真正大声地喊出来，因为我需要知道，这样我才可以解决这个问题！），它就抱怨并死亡。

订阅者如何检测这一点呢？一种方法是将消息依次排列（按顺序给它们编号），并在发布者上使用 HWM。现在，如果订阅者检测到间隙（即编号不连续），它就知道有什么地

方出错了。然后，我们调整 HWM 到“如果你触及这个值，就抱怨并死亡”的级别。

本解决方案存在两个问题。首先，如果我们有很多发布者，怎么将消息依次排列呢？解决的办法是给每个发布者一个唯一 ID 并将它添加到序列号中。其次，如果订阅者使用 ZMQ_SUBSCRIBE 过滤器，他们在理论上就会获得间隙。我们宝贵的顺序将变得一文不值。

一些用例将不使用过滤器，而它们将用编序号的办法来工作。但更一般的解决办法是，发布者对每个消息加时间戳。当订阅者得到一个消息时，它会检查时间，如果收到的时间和时间戳的差值超过某个值，比如说 1 秒，它就执行“抱怨并死亡”操作，这可能会首先向某个操作员控制台发出一个抱怨。

当订阅者有自己的客户端和服务级别协议，并需要保证特定的最大延迟的时候，自杀蜗牛模式效果特别好。终止一个订阅者可能似乎不是一个保证最大延迟的建设性的方式，但它是断言模型。如果现在终止，那么这个问题将被修复。若允许迟到的数据流向下游，问题就可能导致更广泛的破坏并需要较长时间才能被发现。

256 示例 5-10 显示了自杀蜗牛模式的一个简化的例子。

示例5-10：自杀蜗牛（suisnail.c）

```
//
//  自杀蜗牛
//
#include "czmq.h"

//  这是我们的订阅者。它连接到发布服务器并订阅一切。
//  它在消息之间休眠很短的时间，以模拟执行
//  繁重的工作。如果消息延迟超过一秒，它就抱怨。

#define MAX_ALLOWED_DELAY   1000    //  毫秒

static void
subscriber (void *args, zctx_t *ctx, void *pipe)
{
    //  订阅全部东西
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsockopt_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5556");

    //  获取并处理消息
    while (true) {
        char *string = zstr_recv (subscriber);
        printf("%s\n", string);
        int64_t clock;
```

```
        int terms = sscanf (string, "%" PRId64, &clock);
        assert (terms == 1);
        free (string);

        //  自杀蜗牛逻辑
        if (zclock_time () - clock > MAX_ALLOWED_DELAY) {
            fprintf (stderr, "E: subscriber cannot keep up, aborting\n");
            break;
        }
        //  工作1毫秒外加某个随机的时间
        zclock_sleep (1 + randof (2));
    }
    zstr_send (pipe, "gone and died");
}
```

示例 5-11 介绍了我们的发布者任务。它每毫秒发布一个带时间戳的消息给它的 PUB 套接字。

示例5-11：自杀蜗牛（suisnail.c）：发布者任务

```
static void
publisher (void *args, zctx_t *ctx, void *pipe)
{
    //  准备发布者
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");

    while (true) {
        //  发送当前时间（毫秒）给订阅者
        char string [20];
        sprintf (string, "%" PRId64, zclock_time ());
        zstr_send (publisher, string);
        char *signal = zstr_recv_nowait (pipe);
        if (signal) {
            free (signal);
            break;
        }
        zclock_sleep (1);              //  等待1毫秒
    }
}
```

257

主任务（参见示例 5-12）只是启动一个客户端和一个服务器，然后等待客户端发出它已经死掉的信号。

示例5-12：自杀蜗牛（suisnail.c）：主任务

```
int main (void)
```

```
{
    zctx_t *ctx = zctx_new ();
    void *pubpipe = zthread_fork (ctx, publisher, NULL);
    void *subpipe = zthread_fork (ctx, subscriber, NULL);
    free (zstr_recv (subpipe));
    zstr_send (pubpipe, "break");
    zclock_sleep (100);
    zctx_destroy (&ctx);
    return 0;
}
```

自杀蜗牛示例有下列需要注意的事项：

- 这里的消息单纯由当前系统时钟的毫秒数组成。在实际应用程序中，你至少会有一个带时间戳的消息头和带数据的消息体。
- 这个例子的订阅者和发布者是单个进程中的两个线程。在现实中，它们将是独立的进程。在这里使用线程只是为了便于演示。

258

高速订阅者（黑盒模式）

现在，让我们来看一个使我们的订阅者更快的方法。对于发布 - 订阅模式，一个常见的用例是分发大数据流，如来自证券交易所的市场数据。一个典型的设置必须连接到一个证券交易所发布者，取得报价并将它们发送到许多订阅者。如果只有少数订阅者，我们可以使用 TCP。如果订阅者数量较多，我们可能会使用可靠多播，即 PGM。

假设我们的供应方一秒钟平均拥有 100000 个 100 字节的消息。这是将市场数据过滤掉我们不需要发送给订阅者的数据后的典型速度。现在，我们决定记录一天的数据（在 8 小时内，也许有 250 GB），然后将其重播到一个模拟网络（即，一小群订阅者）。虽然对于 ØMQ 应用程序，一秒钟 100KB 的消息是容易做到的，但我们希望用快得多的速度重播。

因此，我们用一批电脑建立了我们的架构，一部分用于发布者，而另一部分用于每个订阅者。这些都是高档的电脑——具有 8 个核心，用于发布者的则有 12 个核心。

当将数据注入我们的订阅者时，我们注意到两件事情：

1. 当我们对一个消息做哪怕是一丁点量的工作时，它也会减慢我们的订阅者，使得他们无法再次赶上发布者的速度。
2. 即使经过仔细的优化和 TCP 调整，我们也会同时在发行方和订阅方触及上限，每秒大约 6M 个消息。

要做的第一件事就是将我们的订阅者分解为一个多线程的设计，使我们可以在一组线程

中处理消息，而在另一组线程中阅读消息。通常情况下，我们不希望用相同的方式处理每个消息。相反，订阅者会过滤一些消息，可能是根据前缀键来过滤。当消息符合某些标准时，订阅者将调用一个工人来处理它。对 ØMQ 而言，这意味着将消息发送到一个工作线程。

因此，订阅者看起来有点像一个队列设备。我们可以使用不同的套接字来连接订阅者和工人。如果假设传输是单向的，并且工人们都是相同的，我们就可以用 PUSH 和 PULL 并将所有路由工作委派给 ØMQ（参见图 5-1）。这是最简单和最快的方法。

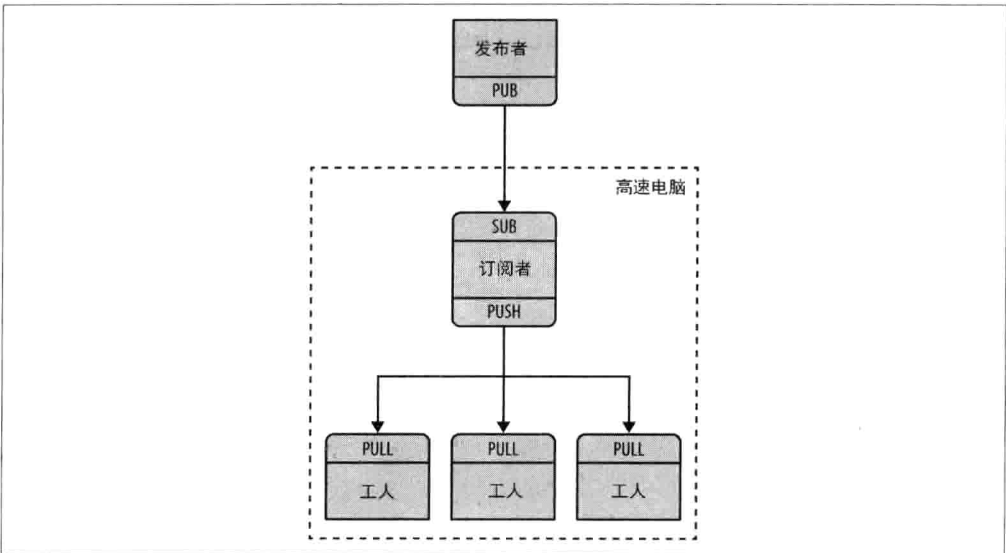


图5-1：简单的黑盒模式

订阅者通过 TCP 或 PGM 与发布者交流。订阅者与它的工人交流，这些都是在同一个进程内，通过 inproc 进行的。 259

我们可以突破这一上限。订阅者线程触及 100%的 CPU，并且因为它是一个线程，所以它不能使用一个以上的核心。单个线程将始终触及上限，无论是每秒 2M、6M，或更多的消息。我们希望把工作拆分为可以并行运行的多个线程。

被许多高性能的产品使用的方法，在这里也能工作，这个方法就是分片（*sharding*）。采用分片，我们把工作拆分为并行和独立的数据流。主题键有一半在一个流中，另一半在另一个流中（参见图 5-2）。我们可以使用许多流，但除非我们有空闲的核心，否则性能不会扩展。

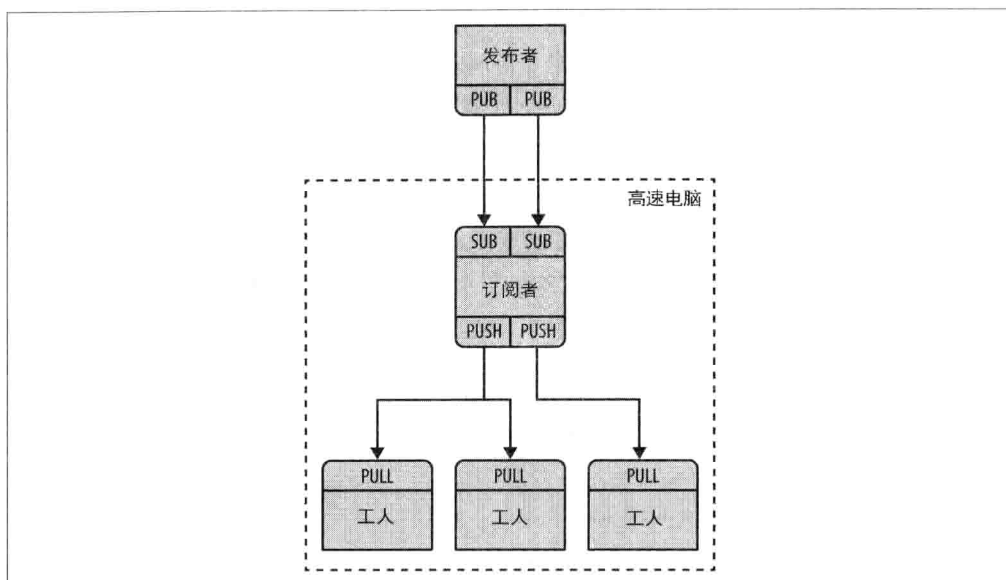


图5-2：疯狂的黑盒模式

260 如果使用两个数据流，就能全速工作，我们将如下配置 ØMQ：

- 两个 I/O 线程，而不是一个。
- 两个网络接口卡（NIC），其中每个订阅者一个。
- 每个 I/O 线程都绑定到特定的网卡。
- 两个订阅者线程，绑定到特定的内核。
- 两个子套接字，其中每个订阅者线程一个。
- 将其余内核分配给工作线程。
- 工作线程连接到这两个订阅者的 PUSH 套接字。

理想的情况下，我们希望在架构中将完全加载的线程数目与核心的数量匹配。当线程开始对核心和 CPU 周期争用时，添加更多线程的成本远远多于好处。

可靠的发布 - 订阅（克隆模式）

作为一个能工作的更大的示例，我们将遇到制作一个可靠的发布 - 订阅架构的问题。我们将分阶段开发这个，我们的目标是允许一组应用程序共享一些共同的状态。以下是我们面临的技术挑战：

261
- 有一大堆客户端应用程序，比如说，几千或几万。

- 这些应用程序将随意加入和离开网络。
- 这些应用程序必须共享一个单一的，最终一致的状态。
- 任何应用程序都可以在任何时间点更新状态。

假定更新量是相当低的，我们没有实时的目标，整个状态可以放入内存。一些似是而非的用例如下所示：

- 由一组云服务器共享的配置。
- 由一群选手分享的某个比赛状态。
- 实时更新并提供给应用程序的汇率数据。

集中式与分散式

我们必须首先做出的决定是是否使用中央服务器来工作，这将使得最终的设计有很大的不同。请权衡下面这些事项：

- 从概念上讲，中央服务器容易理解，因为网络不是自然对称的。使用中央服务器，就避免了发现、绑定与连接等的所有问题。
- 一般情况下，一个完全分布式的架构在技术上更具挑战性，但最后以更简单的协议告终。也就是说，每个节点都必须以正确的方式充当服务器和客户端，这是很难的。如果分布式的架构做得正确，结果会比使用中央服务器简单。我们在第 4 章的自由职业者模式中看到了这一点。
- 中央服务器将成为高容量用例的瓶颈。如果每秒顺序处理数百万规模的消息是必需的，我们应该马上把目标对准分散式架构。
- 集中式架构比分散式架构更容易扩展到多个节点。也就是说，连接 10000 个节点到一台服务器，比将这些节点相互连接更容易实现。

因此，对于克隆模式，我们将发布状态更新的一台服务器和一组表示应用程序的客户端，使二者配合使用。

将状态表示为键 - 值对

我们将分阶段开发克隆模式，每次解决一个问题。首先，让我们来看看如何更新一组客户端的共享状态。我们需要决定如何表示状态，以及更新。最简单可行的格式是一个键 - 值存储，其中一个键 - 值对代表在共享状态中变化的原子单元。

我们在第 2 章看到了一个简单的发布 - 订阅的例子，即天气服务器和客户端。让我们修 262
改服务器，使之发送键 - 值对，而客户端将这些键 - 值对存储在散列表中。这使得我们使用经典的发布 - 订阅模型从一台服务器发送更新到一组客户端（参见图 5-3）。

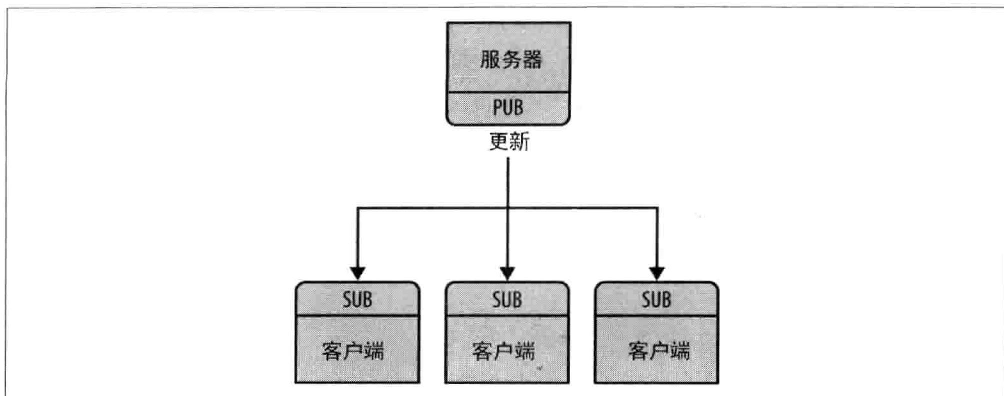


图5-3：发布状态更新

更新要么是一个新的键 - 值对，要么是对现有的键修改后的一个值，要么是一个删除键。现在我们可以假设整个存储区适合容纳在内存中并且应用程序通过键访问它，如通过散列表或字典访问。对于更大的和某种持久性的存储区，我们很可能把状态储存在数据库中，但那与此示例无关。

我们对服务器的第一次尝试如示例 5-13 所示。

示例5-13：克隆服务器，模型一（clonesrv1.c）

```
//
//  克隆服务器 - 模型一
//

//  构建这个源代码而不创建一个库
#include "kvsimple.c"

int main (void)
{
    //  准备上下文和发布者套接字
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");
    zclock_sleep (200);

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

263 while (!zctx_interrupted) {
        //  以键 - 值消息的形式分发
```

```
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_store   (&kvmsg, kvmap);
    }
    printf ("Interrupted\n%d messages out\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

而我们在客户端上的第一次尝试如示例 5-14 所示。

示例5-14：克隆客户端，模型一（clonecli1.c）

```
//
//  克隆客户端 - 模型一
//

//  构建这个源代码而不创建一个库
#include "kvsimple.c"

int main (void)
{
    //  准备上下文和更新套接字
    zctx_t *ctx = zctx_new ();
    void *updates = zsocket_new (ctx, ZMQ_SUB);
    zsockopt_set_subscribe (updates, "");
    zsocket_connect (updates, "tcp://localhost:5556");

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;

    while (true) {
        kvmsg_t *kvmsg = kvmsg_recv (updates);
        if (!kvmsg)
            break;          //  中断
        kvmsg_store (&kvmsg, kvmap);
        sequence++;
    }
    printf ("Interrupted\n%d messages in\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

对于这第一个模型，有下列注意事项：

264
- 所有的繁重工作都是在一个 kvmsg 类中完成的。这个类处理键 - 值消息对象，这是划分为三个帧的多部分 ØMQ 消息：一个键（ØMQ 字符串）、一个序列号（64 位值，以网络字节顺序）和一个二进制的正文（这保存其他一切东西）。
- 服务器生成带有随机四位数的键的消息，它可以让我们模拟一个大而不巨大的散列表（10K 项）。
- 我们不在这个版本中实现删除操作：所有消息都是插入或更新的消息。
- 服务器绑定其套接字之后做了 200 毫秒的停顿。这是为了防止“慢木匠”综合征，其中订阅者当它连接到服务器的套接字时丢失消息。我们将在克隆代码后面的版本中删除它。
- 我们将在代码中使用术语*发布者*和*订阅者*来引用套接字。这将有助于以后我们对多个做不同工作的套接字进行处理。

示例 5-15 显示了 kvmsg 类，是目前能够工作的最简单的形式。

示例5-15：键-值消息类（kvsimple.c）

```
/*  =====================================================================
 *  kvsimple - 示例应用程序的简单的键 - 值消息类
 *  ===================================================================== */

#include "kvsimple.h"
#include "zlist.h"

//  键是短字符串
#define KVMSG_KEY_MAX   255

//  在线路上的消息格式为 4 帧：
//  第 0 帧：键（ØMQ 字符串）
//  第 1 帧：序列号（8 个字节，网络顺序）
//  第 2 帧：正文（二进制大对象）
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_BODY      2
#define KVMSG_FRAMES    3

//  kvmsg 类包含一个单一的键 - 值消息，
//  该消息由 0 个或更多帧的列表组成

struct _kvmsg {
    //  用于每一帧的应答指示器
    int present [KVMSG_FRAMES];
```

```
    // 相应ØMQ消息帧，如果有的话
    zmq_msg_t frame [KVMSG_FRAMES];
    // 复制到安全的C字符串中的键
    char key [KVMSG_KEY_MAX + 1];
};
```

示例 5-16 包含了这个类的构造函数和析构函数的代码。 265

示例5-16：键-值消息类（kvsimple.c）：构造函数和析构函数

```
// 构造函数，为新kvmsg实例取得一个序列号
kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    kvmsg_set_sequence (self, sequence);
    return self;
}

// zhash_free_fn回调辅助程序，做低级别的析构
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        // 销毁消息帧（如果有的话）
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        // 释放对象本身
        free (self);
    }
}

// 析构函数
void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
```

```
        *self_p = NULL;
    }
}
```

recv 方法如示例 5-17 所示，从套接字读取一个键 - 值消息，并返回一个新的 kvmsg 实例。

示例5-17：键-值消息类（kvsimple.c）：接收方法

```
kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
266 kvmsg_t *self = kvmsg_new (0);

    //  读取线路上的所有帧，如果是伪造的则拒绝
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_msg_recv (&self->frame [frame_nbr], socket, 0) == -1) {
            kvmsg_destroy (&self);
            break;
        }
        //  验证多部分组帧
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsockopt_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    return self;
}
```

send 方法（参见示例 5-18）发送一个多帧键 - 值消息到套接字。

示例5-18：键-值消息类（kvsimple.c）：发送方法

```
void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
```

```
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_msg_send (&copy, socket,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}
```

示例 5-19 中的键方法让调用者作为一个固定的字符串和一个 printf 格式的字符串获取和设置消息键。

示例5-19：键-值消息类（kvsimple.c）：键方法

```
char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}

void
```

267

```
kvmsg_fmt_key (kvmsg_t *self, char *format, ...)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}
```

示例 5-20 中的两个方法让调用者获取和设置消息序列号。

示例5-20：键-值消息类（kvsimple.c）：序列方法

```
int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
268     byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}

void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
```

```
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}
```

示例 5-21 中的两个方法让调用者获取和设置消息正文，作为一个固定的字符串和一个 printf 格式的字符串。

示例5-21：键-值消息类（kvsimple.c）：消息正文的方法

```
byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}

void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}

void
kvmsg_fmt_body (kvmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
```

269

```
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}
```

size 方法（参见示例 5-22）返回最近读的消息（如果存在的话）的正文大小。

示例5-22：键-值消息类（kvsimple.c）：大小方法

```
size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}
```

store 方法（参见示例 5-23）将键 - 值消息存储到一个散列映射中，除非键和值都为 null。它置空了 kvmsg 引用，以便该对象是由散列映射，而不是调用者所有。

示例5-23：键-值消息类（kvsimple.c）：存储方法

270

```
void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (self->present [FRAME_KEY]
        &&  self->present [FRAME_BODY]) {
            zhash_update (hash, kvmsg_key (self), self);
            zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
        }
        *self_p = NULL;
    }
}
```

dump 方法如示例 5-24 所示，将键 - 值消息输出到标准错误用于调试和跟踪：

示例5-24：键-值消息类（kvsimple.c）：转储方法

```
void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
```

```
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self));
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd]", size);
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}
```

有一个用来测试类的自测方法，这是很好的做法，这也说明它是如何在应用程序中使用的。我们的自测方法如示例 5-25 所示。

示例5-25：键-值消息类（kvsimple.c）：测试方法

```
int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg:");

    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  测试简单消息的发送和接收
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
```

271

```
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  关闭并销毁所有对象
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

稍后，我们将制作一个更复杂的 kvmsg 类，它将能在实际应用程序中工作。

服务器和客户端维护散列表，但是这第一种模式只有在启动服务器之前启动所有客户端并且客户端永不死机的前提下，才能正常工作。这是非常人为的。

得到带外的快照

所以，现在我们产生了第二个问题：如何处理后期加入的客户端或崩溃后重新启动的客户端。

对于迟到（或恢复中）的客户端，要赶上一台服务器，它必须得到服务器状态的快照。正如我们已经将“消息”简化成是指“一个排序键 - 值对”，我们可以将“状态”简化成是指“一个散列表”。为了获得服务器状态，客户端打开一个 DEALER 套接字，并明确地要求得到它（参见图 5-4）。

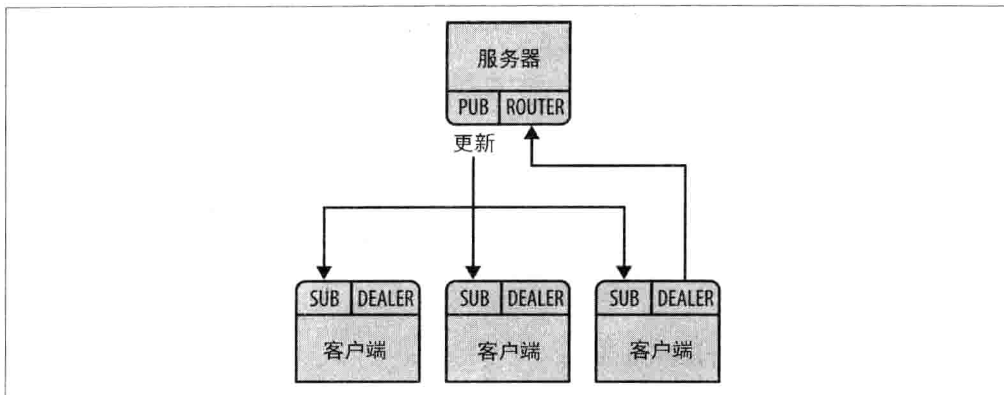


图5-4：状态复制

为了使这能正常工作，我们必须解决计时的问题。得到一个状态的快照将花费一定的时 272 间，如果快照很大，这个时间可能是相当长的。我们要正确地把更新应用到快照上，但服务器不知道何时开始向我们发送更新。一种方法是，开始订阅，得到第一次更新，然后索要“更新 N 的状态”。这就要求服务器为每个更新都存储一个快照，但是，这是不切实际的。

相反，我们将在客户端做同步，如下所示：

- 客户端首先订阅更新，然后执行一个状态请求。这保证了状态将比它拥有的最旧的更新更加新。
- 客户端等待服务器应答状态，并同时对所有更新排队。它会简单地通过不读它们实现这一点：ØMQ 在套接字队列中排列它们。
- 当客户端接收到其状态更新时，它再次开始读取更新。然而，它丢弃早于此状态更新的任何更新（因此，如果状态更新包括多达 200 个更新，客户端将丢弃多达 201 个更新）。
- 客户端然后把更新应用到自己的状态快照上。

它是利用 ØMQ 自己的内部队列的简单模型。我们的克隆服务器模型二如示例 5-26 所示。

示例5-26：克隆服务器，模型二（clonesrv2.c）

```
//
//  克隆服务器 - 模型二
//

//  构建这个源代码而不创建一个库
#include "kvsimple.c"
```

273

```
static int s_send_single (const char *key, void *data, void *args);
static void state_manager (void *args, zctx_t *ctx, void *pipe);

int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");

    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    //  启动状态管理器并等待同步信号
    void *updates = zthread_fork (ctx, state_manager, NULL);
    free (zstr_recv (updates));

    while (!zctx_interrupted) {
        //  以键 - 值消息的形式分发
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_send     (kvmsg, updates);
        kvmsg_destroy (&kvmsg);
    }
    printf (" Interrupted\n%d messages out\n", (int) sequence);
    zctx_destroy (&ctx);
    return 0;
}

//  一个键 - 值快照的路由信息
typedef struct {
    void *socket;           //  要发往的 ROUTER 套接字
    zframe_t *identity;     //  请求了状态的节点身份
} kvroute_t;

//  将一个状态快照键 - 值对发送到一个套接字
//  散列条目数据是 kvmsg 对象，准备发送
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  首先发送接收者身份
    zframe_send (&kvroute->identity,
```

```
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}
```

状态管理任务，如示例 5-27 所示，维护状态并处理来自客户端的快照请求。 274

示例5-27：克隆服务器，模型二（clonesrv2.c）：状态管理

```
static void
state_manager (void *args, zctx_t *ctx, void *pipe)
{
    zhash_t *kvmap = zhash_new ();

    zstr_send (pipe, "READY");
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");

    zmq_pollitem_t items [] = {
        { pipe, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    int64_t sequence = 0;       //  当前快照版本号
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1 && errno == ETERM)
            break;              //  上下文已被关闭

        //  应用来自主线程的状态更新
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (pipe);
            if (!kvmsg)
                break;          //  中断
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        //  执行状态快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  中断

            //  请求在消息的第 2 帧中
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
```

```
            else {
                printf ("E: bad request, aborting\n");
                break;
            }
            //  将状态快照发送给客户端
            kvroute_t routing = { snapshot, identity };

            //  对于 kvmap 中的每个条目，发送 kvmsg 给客户端
            zhash_foreach (kvmap, s_send_single, &routing);

275         //  现在发送带有序号的 END 消息
            printf ("Sending state shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    zhash_destroy (&kvmap);
}
```

我们克隆的客户端模型二如示例 5-28 所示。

示例5-28：克隆客户端，模型二（clonecli2.c）

```
//
//  克隆客户端 - 模型二
//

//  构建这个源代码而不创建一个库
#include "kvsimple.c"

int main (void)
{
    //  准备上下文和订阅者
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsockopt_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5557");

    zhash_t *kvmap = zhash_new ();

    //  获取状态快照
```

```
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (true) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  中断
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("Received snapshot=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  完成
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    //  现在应用待定的更新，丢弃序号不对的消息
    while (!zctx_interrupted) {
        kvmsg_t *kvmsg = kvmsg_recv (subscriber);                          276
        if (!kvmsg)
            break;          //  中断
        if (kvmsg_sequence (kvmsg) > sequence) {
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

这两个程序中有下列需要注意的事项：

- 服务器使用两个任务。一个线程（随机地）产生更新，并且将这些发送给主 PUB 套接字，而另一个线程处理 ROUTER 套接字上的状态请求。两个跨 PAIR 套接字的通信通过一个 inproc 连接进行。
- 客户端是非常简单的。在 C 语言中它大约由 50 行代码组成。很多繁重的工作是在 kvmsg 类中完成的。即便如此，基本克隆模式比它起初看上去更容易实现。
- 我们不使用任何花哨的东西来序列化状态。散列表保存了一组 kvmsg 对象，而服务器将这些作为一个批处理的消息发送给请求状态的客户端。如果多个客户端同时请求状态，那么每个客户端都会得到一个不同的快照。
- 我们假设客户端有且只有一个服务器要交流。服务器必须在运行，我们并不试图解决如果服务器崩溃会发生的问题。

眼下,这两个程序没有做任何实际的工作,但它们正确地同步状态。这是一个整洁的例子,它说明了如何搭配不同的模式：PAIR-PAIR、PUB-SUB 和 ROUTER-DEALER。

重新发布来自客户端的更新

在我们的第二个模型中,对键-值存储的改变来自服务器本身。这是一个有用的集中模型,例如，如果我们有一个中心配置文件要分发，且在每个节点上都带有该配置文件的本地缓存，那么就适用这个模型。一个更有趣的模型是从客户端获取更新，而不是从服务器获取更新。服务器也因此成为一个无状态的代理，这带给了我们几个好处：

- 不用太担心服务器的可靠性。如果它崩溃，我们就可以开始一个新的实例，并给它提供新值。
- 277 可以使用键-值存储在活动节点之间共享知识。

为了将来自客户端的更新发送回服务器，我们可以使用各种套接字的模式。最简单可行的解决方案是一个 PUSH-PULL 组合（参见图 5-5）。

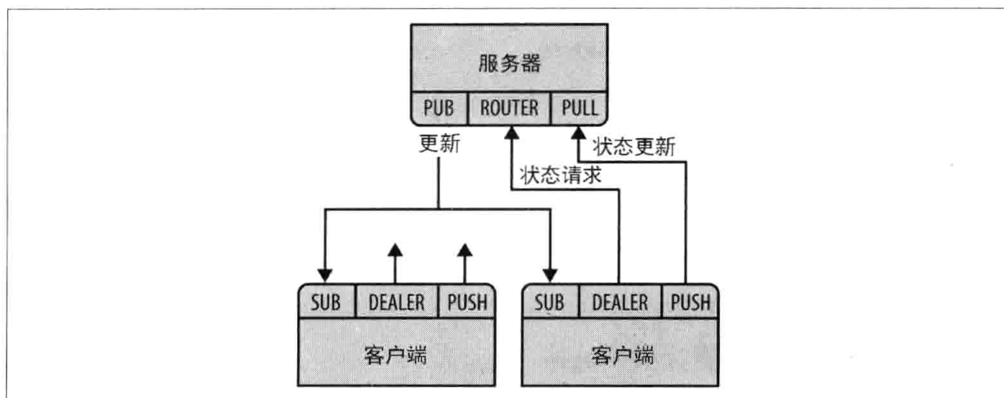


图5-5：重新发布更新

为什么我们不能让客户端直接相互发布更新呢？虽然这会减少延迟，但这将失去一致性的保证。如果允许按照谁收到它们的顺序来改变更新的顺序，你就不能得到一致的共享状态。如果两个客户端同时进行更改，但更改的键不同，就没有混淆。但是，如果两个客户端试图在大致相同的时间改变相同的键，它们将最终获得它不同概念的值。

当更改同时在多个地方发生时，有几个策略可用来获得一致性。我们将使用集中化所有变化的方法。不管客户端进行更改的精确时间如何，它们都通过服务器推送，该服务器根据它得到的更新顺序，对更新强加单个的序号。

通过居中调解所有更改，服务器还可以为全部更新添加一个唯一的序号。有了独特的序号，客户端就可以检测到令人讨厌的故障、网络拥塞和队列溢出。如果客户端发现其传入消息流有一个洞，就可以采取行动。客户端与服务器联系并索要丢失的消息，这似乎是合理的，但在实践中是没有用的。如果有漏洞，它们是由网络压力引起的，而对网络增加更多的压力，将会使事情变得更糟。所有的客户端能做的就是，提醒它的订阅者，它"无法继续"，停下来，并且不要重新启动，直到有人手动检查出了问题的原因。

在我们的第三个模型中，将在客户端生成状态更新。服务器代码如示例 5-29 所示。

示例5-29：克隆服务器，模型三（clonesrv3.c） 278

```
//
//  克隆服务器，模型三
//

//  构建这个源代码而不创建一个库
#include "kvsimple.c"

//  一个键 - 值快照的路由信息
typedef struct {
    void *socket;           //  要发往的 ROUTER 套接字
    zframe_t *identity;     //  请求了状态的节点身份
} kvroute_t;

//  将一个状态快照键 - 值对发送到一个套接字
//  散列条目数据是 kvmsg 对象，准备发送
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  首先发送接收者身份
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}

int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
```

```
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");
```

主任务的主体，如示例 5-30 所示，收集来自客户端的更新并将它们发布回客户端。

示例5-30：克隆服务器，模型三（clonesrv3.c）：主任务的主体

279

```
    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  应用发自客户端的状态更新
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  中断
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: publishing update %5d\n", (int) sequence);
        }
        //  执行状态快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  中断

            //  请求在消息的第 2 帧中
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: bad request, aborting\n");
                break;
            }
            //  将状态快照发送给客户端
            kvroute_t routing = { snapshot, identity };

            //  对于 kvmap 中的每个条目，发送 kvmsg 给客户端
            zhash_foreach (kvmap, s_send_single, &routing);
```

```
            // 现在发送带有序号的 END 消息
            printf ("I: sending shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    printf ("Interrupted\n%d messages handled\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}
```

客户端的模型三的代码如示例 5-31 到示例 5-33 所示。

示例5-31：克隆客户端，模型三（clonecli3.c） 280

```
//
// 克隆客户端 - 模型三
//

// 构建这个源代码而不创建一个库
#include "kvsimple.c"

int main (void)
{
    // 准备上下文和订阅者
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsockopt_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5557");
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));
```

首先，我们请求一个状态快照，如示例 5-32 所示。

示例5-32：克隆客户端，模型三（clonecli3.c）：获得一个状态快照

```
zstr_send (snapshot, "ICANHAZ?");
while (true) {
    kvmsg_t *kvmsg = kvmsg_recv (snapshot);
    if (!kvmsg)
        break;          //  中断
    if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
        sequence = kvmsg_sequence (kvmsg);
        printf ("I: received snapshot=%d\n", (int) sequence);
        kvmsg_destroy (&kvmsg);
        break;          //  完成
    }
    kvmsg_store (&kvmsg, kvmap);
}
```

然后我们等待来自服务器的更新，并且每隔一段时间，发送一个随机的键值更新到服务器，如示例 5-33 所示。

示例5-33：克隆客户端，模型三（clonecli3.c）：处理状态更新

```
int64_t alarm = zclock_time () + 1000;
while (!zctx_interrupted) {
    zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 } };
    int tickless = (int) ((alarm - zclock_time ()));
    if (tickless < 0)
        tickless = 0;
281 int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
    if (rc == -1)
        break;              //  上下文已被关闭

    if (items [0].revents & ZMQ_POLLIN) {
        kvmsg_t *kvmsg = kvmsg_recv (subscriber);
        if (!kvmsg)
            break;          //  中断

        //  丢弃序号不对的 kvmsgs ，包括信号检测
        if (kvmsg_sequence (kvmsg) > sequence) {
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: received update=%d\n", (int) sequence);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    //  如果我们超时了，就生成一个随机 kvmsg
    if (zclock_time () >= alarm) {
```

```
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
    }
    printf ("Interrupted\n%d messages in\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

第三种设计有下列要注意的事项：

- 服务器已经合并成单个的任务。它管理一个 PULL 套接字用于传入的更新，一个 ROUTER 套接字用于状态请求，以及一个 PUB 套接字用于传出的更新。
- 客户端使用一个简单的无滴答定时器每秒一次向服务器发送一个随机更新。在实际的实现中，我们将从应用程序代码驱动更新。

处理子树

随着客户端数量的增长，我们的共享存储区的规模也将增长。最终，将一切都发送到每一个客户端就会变得不再可行。这是使用发布 - 订阅模式的经典故事：当你有非常少的客户端时，可以发送每个消息给所有客户端，但随着架构的成长，这将变得效率低下。282
客户端都有特定的不同领域。

所以，即使用一个共享的存储区来处理，有些客户端会希望只与那个存储区的一部分一同工作，我们将存储区的这一部分称为子树（*subtree*）。当客户端执行一个状态请求时，它必须请求子树，并且当它订阅更新时，必须指定相同的子树。

树的常见语法有两种。一种是路径的层次结构（*path hierarchy*），而另一种是主题树（*topic tree*）。它们看起来像下面这样：

- 路径的层次结构：*/some/list/of/paths*
- 主题树：*some.list.of.topics*

我们将使用路径的层次结构并扩充我们的客户端和服务器，使客户端可以处理一个子树。一旦了解如何处理一个子树，你就可以自己扩充这个例子来处理多个子树，如果你的用例需要它。

示例 5-34 显示了实现子树的服务器，一个在模型三的基础上的小变体。

示例5-34：克隆服务器，模型四（clonesrv4.c）

```
//
//  克隆服务器 - 模型四
//

//  构建这个源代码而不创建一个库
#include "kvsimple.c"

//  一个键 - 值快照的路由信息
typedef struct {
    void *socket;           //  要发往的 ROUTER 套接字
    zframe_t *identity;     //  请求了状态的节点身份
    char *subtree;          //  客户端子树的规范
} kvroute_t;

//  将一个状态快照键 - 值对发送到一个套接字
//  散列条目数据是 kvmsg 对象，准备发送
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  首先发送接收者身份
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
283 }
    return 0;
}

//  主任务与 clonesrv3 相同，除了它处理子树的部分
...
            //  请求在消息的第 2 帧中
            char *request = zstr_recv (snapshot);
            char *subtree = NULL;
            if (streq (request, "ICANHAZ?")) {
                free (request);
                subtree = zstr_recv (snapshot);
            }
...
            //  将状态快照发送到客户端
```

```
            kvroute_t routing = { snapshot, identity, subtree };
...
            //  现在发送带序列号的 END 消息
            printf ("I: sending shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
...
```

相应的客户端代码如示例 5-35 所示。

示例5-35：克隆客户端，模型四（clonecli4.c）

```
//
//  克隆客户端 - 模型四
//

//  构建这个源代码而不创建一个库
#include "kvsimple.c"

//  这个客户端与 clonecli3 相同，除了处理子树的部分
#define SUBTREE "/client/"
...
    zsocket_connect (subscriber, "tcp://localhost:5557");
    zsockopt_set_subscribe (subscriber, SUBTREE);
...
    //  首先请求一个状态快照
    int64_t sequence = 0;
    zstr_sendm (snapshot, "ICANHAZ?");
    zstr_send  (snapshot, SUBTREE);
...
        //  如果我们超时了，就生成一个随机 kvmsg
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%s%d", SUBTREE, randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
...
```

284

临时值

临时值是指除非定期刷新，否则自动失效的值。如果你考虑被用于注册服务的克隆模式，则临时值将让你能够使用动态值。当一个节点加入网络时，发布其地址，并定期刷新这个值。如果节点死机，其地址最终被删除。

对临时值通常的抽象是将它们附加到一个会话，并在会话结束时删除它们。在克隆模式中，会话将通过客户端进行定义，并在客户端死掉时结束。一个更简单的方法是将一个生存时间（*TTL*）附加到临时值，服务器使用它来将还没有被按时刷新的值设置为过期。

任何时候尽可能采用的一个好的设计原则是不去发明不是绝对必要的概念。如果我们有大量临时值，会话就会提供更好的性能。如果使用了少量的临时值，那么对每一个临时值设置一个 TTL 是不错的。如果有大量的临时值，将其连接到会话并批量使它们过期会更有效。这不是我们在这个阶段所面对的问题，而且我们可能永远不会面对这个问题，所以会话在这里是不重要的。

现在，我们将实现临时值。首先，需要一种方法来将 TTL 编码在键 - 值消息中。我们可以添加一帧，但用多个 ØMQ 帧来保存多个属性的问题是，每次要添加一个新的属性时，都要改变消息的结构。这样就破坏了兼容性。所以，让我们在消息中添加一个属性帧，并编写代码来从中获取和放置属性值。

接下来，我们需要一种方式来“删除此值”。到现在为止，服务器和客户端都总是一味地插入或更新新的值到它们的散列表中。我们会规定，如果值是空的，这就意味着“删除此键”。

示例 5-36 显示了 kvmsg 类的一个更完整的版本，它实现了一个属性帧（并增加了一个 UUID 帧，我们在以后将需要它）。它也通过从散列中删除键来处理空值，如果有必要的话。

示例5-36：键-值消息类：完整版（kvmsg.c）

```
/*  =====================================================================
 *  kvmsg - 示例应用程序的键 - 值消息类 kvmsg
285 *  ===================================================================== */

#include "kvmsg.h"
#include <uuid/uuid.h>
#include "zlist.h"

//  键是短字符串
#define KVMSG_KEY_MAX    255

//  在线路上的消息格式为 4 帧：(译者注：应该是 5 帧)
```

```
//  第 0 帧：键（ØMQ 字符串）
//  第 1 帧：序列号（8 个字节，网络顺序）
//  第 2 帧：uuid（blob，16 字节）
//  第 3 帧：属性（ØMQ 字符串）
//  第 4 帧：正文（二进制大对象）
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_UUID      2
#define FRAME_PROPS     3
#define FRAME_BODY      4
#define KVMSG_FRAMES    5

//  类结构
struct _kvmsg {
    //  用于每一帧的存在性指示器
    int present [KVMSG_FRAMES];
    //  相应的 ØMQ 消息帧，如果有的话
    zmq_msg_t frame [KVMSG_FRAMES];
    //  复制到安全的 C 字符串中的键
    char key [KVMSG_KEY_MAX + 1];
    //  属性列表，以“名称=值”字符串的形式表示
    zlist_t *props;
    size_t props_size;
};
```

示例 5-37 中的两个辅助方法分别把属性列表序列化到一个消息帧，和从消息帧序列化到属性列表。

示例5-37：键-值消息类，完整版（kvmsg.c）：属性编码

```
static void
s_encode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    if (self->present [FRAME_PROPS])
        zmq_msg_close (msg);

    zmq_msg_init_size (msg, self->props_size);
    char *prop = zlist_first (self->props);
    char *dest = (char *) zmq_msg_data (msg);
    while (prop) {
        strcpy (dest, prop);
        dest += strlen (prop);
        *dest++ = '\n';
        prop = zlist_next (self->props);
```

286

```
    }
    self->present [FRAME_PROPS] = 1;
}

static void
s_decode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    self->props_size = 0;
    while (zlist_size (self->props))
        free (zlist_pop (self->props));

    size_t remainder = zmq_msg_size (msg);
    char *prop = (char *) zmq_msg_data (msg);
    char *eoln = memchr (prop, '\n', remainder);
    while (eoln) {
        *eoln = 0;
        zlist_append (self->props, strdup (prop));
        self->props_size += strlen (prop) + 1;
        remainder -= strlen (prop) + 1;
        prop = eoln + 1;
        eoln = memchr (prop, '\n', remainder);
    }
}
```

类的构造函数和析构函数如示例 5-38 所示。

示例5-38：键-值消息类，完整版（kvmsg.c）：构造函数和析构函数

```
//  构造函数，为新 kvmsg 实例取得一个序列号
kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    self->props = zlist_new ();
    kvmsg_set_sequence (self, sequence);
    return self;
}

//  zhash_free_fn 回调辅助程序，做低级别的析构
void
kvmsg_free (void *ptr)
{
    if (ptr) {
```

```
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  销毁消息帧（如果有的话）
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)    287
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        //  销毁属性列表
        while (zlist_size (self->props))
            free (zlist_pop (self->props));
        zlist_destroy (&self->props);

        //  释放对象本身
        free (self);
    }
}

//  析构函数
void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}
```

示例 5-39 中的 recv 方法从套接字读取一个键 - 值消息，并返回一个新的 kvmsg 实例。

示例5-39：键-值消息类，完整版（kvmsg.c）：recv方法

```
kvmsg_t *
kvmsg_recv (void *socket)
{
    //  这个方法几乎与 kvsimple 没有区别
...
    if (self)
        s_decode_props (self);
    return self;
}


//  ---------------------------------------------------------------------
//  发送键 - 值消息到套接字，任何空帧都按原样发送

void
```

```
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    s_encode_props (self);
    //  方法的其余部分与 kvsimple 没有区别
...
```

288

dup 方法（参见示例 5-40）复制一个 kvmsg 实例，并返回新的实例。

示例5-40：键-值消息类，完整版（kvmsg.c）：dup方法

```
kvmsg_t *
kvmsg_dup (kvmsg_t *self)
{
    kvmsg_t *kvmsg = kvmsg_new (0);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr]) {
            zmq_msg_t *src = &self->frame [frame_nbr];
            zmq_msg_t *dst = &kvmsg->frame [frame_nbr];
            zmq_msg_init_size (dst, zmq_msg_size (src));
            memcpy (zmq_msg_data (dst),
                    zmq_msg_data (src), zmq_msg_size (src));
            kvmsg->present [frame_nbr] = 1;
        }
    }
    kvmsg->props_size = zlist_size (self->props);
    char *prop = (char *) zlist_first (self->props);
    while (prop) {
        zlist_append (kvmsg->props, strdup (prop));
        prop = (char *) zlist_next (self->props);
    }
    return kvmsg;
}

//  key、sequence、body 和 size 方法和 kvsimple 中的相同
...
```

示例 5-41 中的方法为键 - 值消息获取和设置 UUID。

示例5-41：键-值消息类，完整版（kvmsg.c）：UUID方法

```
byte *
kvmsg_uuid (kvmsg_t *self)
{
    assert (self);
```

```
    if (self->present [FRAME_UUID]
    &&  zmq_msg_size (&self->frame [FRAME_UUID]) == sizeof (uuid_t))
        return (byte *) zmq_msg_data (&self->frame [FRAME_UUID]);
    else
        return NULL;
}

//  将 UUID 设置为一个随机生成的值
void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
    uuid_generate (uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid));
    memcpy (zmq_msg_data (msg), uuid, sizeof (uuid));
    self->present [FRAME_UUID] = 1;
}
```

289

示例 5-42 中的方法获取和设置指定的消息属性。

示例5-42：键-值消息类，完整版（kvmsg.c）：属性方法

```
//  获取消息属性，如果没有定义这样的属性，则返回 ""
char *
kvmsg_get_prop (kvmsg_t *self, char *name)
{
    assert (strchr (name, '=') == NULL);
    char *prop = zlist_first (self->props);
    size_t namelen = strlen (name);
    while (prop) {
        if (strlen (prop) > namelen
        &&  memcmp (prop, name, namelen) == 0
        &&  prop [namelen] == '=')
            return prop + namelen + 1;
        prop = zlist_next (self->props);
    }
    return "";
}


//  设置消息属性。属性名不能包含“=”
//  属性值的最大长度是 255 个字符。
void
```

```
kvmsg_set_prop (kvmsg_t *self, char *name, char *format, ...)
{
    assert (strchr (name, '=') == NULL);

    char value [255 + 1];
    va_list args;
    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);

    //  分配 name=value 字符串
    char *prop = malloc (strlen (name) + strlen (value) + 2);

    //  移除现存的属性（如果有的话）
    sprintf (prop, "%s=", name);
290 char *existing = zlist_first (self->props);
    while (existing) {
        if (memcmp (prop, existing, strlen (prop)) == 0) {
            self->props_size -= strlen (existing) + 1;
            zlist_remove (self->props, existing);
            free (existing);
            break;
        }
        existing = zlist_next (self->props);
    }
    //  添加新的 name=value 属性字符串
    strcat (prop, value);
    zlist_append (self->props, prop);
    self->props_size += strlen (prop) + 1;
}
```

store 方法（参见示例 5-43）将键 - 值消息存储到一个散列映射，除非键和值都为 null。它置空了 kvmsg 引用，以便该对象是由散列映射所有，而不是调用者所有。

示例5-43：键-值消息类，完整版（kvmsg.c）：存储方法

```
void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (kvmsg_size (self)) {
            if (self->present [FRAME_KEY]
```

```
            && self->present [FRAME_BODY]) {
                zhash_update (hash, kvmsg_key (self), self);
                zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
            }
        }
        else
            zhash_delete (hash, kvmsg_key (self));

        *self_p = NULL;
    }
}
```

dump 方法（参见示例 5-44）用对消息属性的支持扩展了 kvsimple 实现。

示例5-44：键-值消息类，完整版（kvmsg.c）：转储方法

```
void
kvmsg_dump (kvmsg_t *self)
{
...
        fprintf (stderr, "[size:%zd]", size);
        if (zlist_size (self->props)) {
            fprintf (stderr,"[");
            char *prop = zlist_first (self->props);
            while (prop) {
                fprintf (stderr, "%s;", prop);
                prop = zlist_next (self->props);
            }
            fprintf (stderr, "]");
        }
...
```

291

selftest（译者注：从代码看没有 self 字样）方法，如示例 5-45 所示，与 kvsimple 中的是相同的，它还添加了对 kvmsg 的 UUID 和属性功能的支持。

示例5-45：键-值消息类，完整版（kvmsg.c）：测试方法

```
int
kvmsg_test (int verbose)
{
...
    //  测试简单消息的发送与接收
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
```

```
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  测试带有属性的消息的发送和接收
    kvmsg = kvmsg_new (2);
    kvmsg_set_prop (kvmsg, "prop1", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value2");
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_destroy (&kvmsg);

    kvmsg = kvmsg_recv (input);
292 if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    kvmsg_destroy (&kvmsg);
...
```

模型五的客户端与模型四几乎是相同的。它现在采用了完整的 kvmsg 类，并在每个消息上设置随机 ttl 属性（以秒为单位）：

```
    kvmsg_set_prop (kvmsg, "ttl", "%d", randof (30));
```

使用反应器

到现在为止，我们已经在服务器中使用了轮询循环。在服务器的下一模型中，我们切换到使用反应器。在 C 语言中，我们使用 CZMQ 的 zloop 类。使用反应器使得代码更加冗长，但更容易理解和构建，因为服务器的每一个部件都由一个单独的反应器处理程序。

我们使用单线程，并把一个服务器对象传递给反应器处理程序。也可以将服务器组织为多线程，每个线程处理一个套接字或定时器，但当线程不必共享数据时，它们工作得更好。在这种情况下，所有的工作都围绕着服务器的散列映射进行，所以一个线程更简单。

存在三个反应器处理程序：

- 一个用于处理在 ROUTER 套接字上传入的快照请求。
- 一个用于处理来自客户端的更新，在 PULL 套接字上传入。
- 一个用于将已超过其 TTL 的临时值置为过期。

克隆服务器的模型五的代码如示例 5-46 所示。

示例5-46：克隆服务器，模型五（clonesrv5.c）

```
//
//  克隆服务器 - 模型五
//

//  构建这个源代码而不创建一个库
#include "kvmsg.c"

//  zloop 反应器处理程序
static int s_snapshots  (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_collector  (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_flush_ttl  (zloop_t *loop, zmq_pollitem_t *poller, void *args);

//  服务器是由这些属性定义的
typedef struct {
    zctx_t *ctx;                //  上下文包装器
    zhash_t *kvmap;             //  键 - 值存储区
    zloop_t *loop;              //  zloop 反应器                                  293
    int port;                   //  我们正在上面工作的主端口
    int64_t sequence;           //  已经完成的更新的数量
    void *snapshot;             //  处理快照请求
    void *publisher;            //  发布更新到客户端
    void *collector;            //  收集来自客户端的更新
} clonesrv_t;


int main (void)
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));

    self->port = 5556;
    self->ctx = zctx_new ();
```

```
    self->kvmap = zhash_new ();
    self->loop = zloop_new ();
    zloop_set_verbose (self->loop, FALSE);

    //  设置克隆服务器套接字
    self->snapshot  = zsocket_new (self->ctx, ZMQ_ROUTER);
    zsocket_bind (self->snapshot,  "tcp://*:%d", self->port);
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    self->collector = zsocket_new (self->ctx, ZMQ_PULL);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  将处理程序注册到反应器
    zmq_pollitem_t poller = { 0, 0, ZMQ_POLLIN };
    poller.socket = self->snapshot;
    zloop_poller (self->loop, &poller, s_snapshots, self);
    poller.socket = self->collector;
    zloop_poller (self->loop, &poller, s_collector, self);
    zloop_timer (self->loop, 1000, 0, s_flush_ttl, self);

    //  反应器一直运行，直到进程中断
    zloop_start (self->loop);

    zloop_destroy (&self->loop);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);
    return 0;
}
```

我们通过发送快照数据给请求它的客户端处理“ICANHAZ?”请求，如示例 5-47 所示。

示例5-47：克隆服务器，模型五（clonesrv5.c）：发送快照

```
//  一个键 - 值快照的路由信息
typedef struct {
294 void *socket;           //  要发往的 ROUTER 套接字
    zframe_t *identity;     //  请求了状态的节点身份
    char *subtree;          //  客户端子树规范
} kvroute_t;

//  我们为散列表中的每个键 - 值对调用此函数
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
```

```
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        zframe_send (&kvroute->identity,    //  选择接收者
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}
```

示例 5-48 显示了用于快照套接字的反应器处理程序，它只是接受“ICANHAZ?”请求，并用以 KTHXBAI 消息结束的一个状态快照来应答。

示例5-48：克隆服务器，模型五（clonesrv5.c）：快照处理程序

```
static int
s_snapshots (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (poller->socket);
    if (identity) {
        //  请求在消息的第 2 帧中
        char *request = zstr_recv (poller->socket);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (poller->socket);
        }
        else
            printf ("E: bad request, aborting\n");

        if (subtree) {
            //  发送状态套接字给客户端
            kvroute_t routing = { poller->socket, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  现在发送带有序号的 END 消息
            zclock_log ("I: sending shapshot=%d", (int) self->sequence);
            zframe_send (&identity, poller->socket, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, poller->socket);
            kvmsg_destroy (&kvmsg);
            free (subtree);
```

295

```
        }
        zframe_destroy(&identity);
    }
    return 0;
}
```

我们在存储每个更新时都使用新的序列号，并且如果有必要，还包括一个生存时间，如示例 5-49 所示。我们立即在发布者套接字上发布更新。

示例5-49：克隆服务器，模型五（clonesrv5.c）：收集更新

```
static int
s_collector (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (poller->socket);
    if (kvmsg) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
        if (ttl)
            kvmsg_set_prop (kvmsg, "ttl",
                "%" PRId64, zclock_time () + ttl * 1000);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing update=%d", (int) self->sequence);
    }
    return 0;
}
```

每隔一段时间，我们就刷新已过期的临时值（参见示例 5-50）。这在非常大的数据集上可能是缓慢的。

示例5-50：克隆服务器，模型五（clonesrv5.c）：刷新临时值

```
//  如果键 - 值对已经过期，删除它并把这个事实发布给监听客户端
static int
s_flush_single (const char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
```

```
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing delete=%d", (int) self->sequence);
    }
    return 0;
}

static int
s_flush_ttl (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    if (self->kvmap)
        zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}
```
296

在双星模式中添加可靠性

到目前为止，我们已经探索的克隆模式相对都比较简单。但是，现在我们要进入令人不快的复杂领域了。你应该明白一个事实，即实现“可靠”的消息传递是如此复杂，以至于你在深入进去之前总是要问，“我们是否真的需要这个呢？”如果你能从不可靠，或“足够好”的可靠性中脱身，就可以在成本和复杂性方面取得一个巨大的胜利。当然，你可能偶尔会丢失一些数据，但可靠性往往是一个很好的权衡。

当用最后一个模型工作时，你会停止并重新启动服务器。它可能看起来像在恢复，但当然它正将更新应用到一个空的状态，而不是正确的当前状态上。任何加入网络的新客户端只会获得最新的更新，而不是完整的更新历史纪录。

我们要的是一种让服务器从被清除或崩溃中恢复的方法。我们还需要提供备份，以防服务器停止运行任意时间。当人们要求“可靠性”时，要求他们列出他们要处理的故障。在我们的例子中，它们是：

- 服务器进程崩溃，并自动或手动重新启动。这个进程丢失了它的状态，并且必须从某个地方把它找回来。
- 服务器机器死机并离线一段显著的时间。客户端必须切换到在某处的备用服务器。
- 服务器进程或机器被从网络中断开，比如交换机死机或数据中心被破坏。它可能在某些时候回来，但在此期间客户端需要一台备用服务器。

我们的第一步是添加第二个服务器。可以使用来自第 6 章的双星模式将这些组织成一个主服务器和一个备用服务器。双星模式是一个反应器模式，所以，我们已经将最后一个服务器模式重构为反应器的形式是有用的。 297

我们需要确保，即使主服务器崩溃也不会丢失更新。最简单的方法是将它们同时发送到两台服务器。然后,备用服务器可以充当一个客户端,并通过接收更新保持它的状态同步,正如所有的客户端所做的。它也将获得来自客户端的新的更新。它还不可以将这些存储在其散列表中，但它可以保持它们一段时间。

因此，模型六在模型五的基础上引入了以下更改：

- 使用一个发布 - 订阅流，而不是一个推送 - 提取流来把客户端的更新发送到服务器。这负责同时扇出更新到两台服务器。否则，我们不得不使用两个 DEALER 套接字。
- 把信号检测增加到服务器的更新中（对客户端），使得客户端可以检测到主服务器已经死机这种情况。然后，它可以切换到备份服务器。
- 使用双星 bstar 反应器类连接两台服务器。双星依赖于客户端通过做出明确的请求到它们认为“活动”的服务器来“投票”，我们将使用快照请求作为表决机制。
- 通过添加一个 UUID 字段使得所有更新消息都可唯一辨认。客户端生成此字段，而服务器在重新发布的更新中将它传播回来。
- 被动服务器保留更新的“待定清单”，该清单保存的是它已经从客户端收到但尚未从活动服务器收到的更新，以及它从活动服务器收到，但尚未从客户端收到的更新。该清单是从旧到新排序的，所以很容易移除最前面的更新。

将客户端逻辑设计为一个有限状态机，这是非常有用的。客户端循环遍历如下三种状态：

1. 客户端打开并连接到它的套接字，然后从第一个服务器请求一个快照。为了避免请求的风暴，它会对任何给定的服务器只请求两次。可能会丢失一个请求，这是运气不好。而两个请求都丢失则是粗心大意。
2. 客户端从当前服务器等待应答（快照数据），如果它得到了应答，就把它存储起来。如果在某个超时时间内无应答，它会自动切换到下一个服务器。
3. 当客户端已经得到它的快照，它会等待并处理更新。同样，如果它在某个超时时间内未从服务器收到任何东西，它会自动切换到下一个服务器。

客户端永远循环。这与在启动或故障转移期间，一些客户端可能会试图与主服务器交流，而其他客户端都试图与备份服务器交流的过程很相似。可以指望双星状态机很准确地处理
298 这种情况（参见图 5-6）。想要证明软件正确，这是很难的，相反，我们采取的办法是不断攻击它，直到无法证明它错了为止。

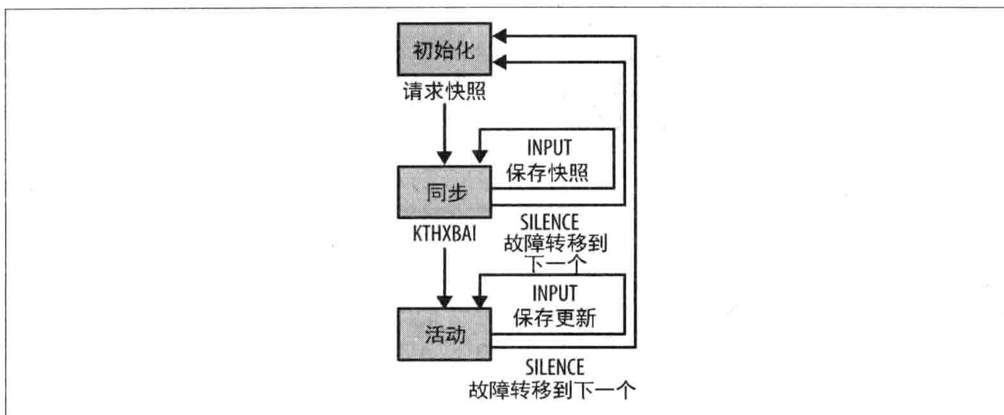


图5-6：克隆客户端的有限状态机

故障转移以如下方式发生：

- 客户端检测到主服务器不再发送检测信号，并得出结论，它已经死机了。客户端连接到备用服务器并请求一个新的状态快照。
- 备用服务器开始从客户端接收快照请求，并检测到主服务器已经离开了，所以它作为主服务器身份接管。
- 备用服务器将其待定清单应用到自身的散列表，然后开始处理状态快照请求。

当主服务器恢复在线时，它将会：

- 以被动服务器身份启动，并作为一个克隆的客户端连接到备用服务器。
- 开始通过其 SUB 套接字从客户端接收更新。

我们做了一些假设：

- 至少有一台服务器将继续运行。如果两台服务器全都崩溃，我们就失去了所有服务器的状态，也就没有办法恢复了。
- 多个客户端不会同时更新相同的散列键。客户端更新将以不同的顺序到达两个服务器。因此，备用服务器从它的待定清单应用更新，其顺序可能与主服务器将执行或
已执行的更新顺序不同。来自一个客户端的更新总是会以相同的顺序到达两台服务 299
器，所以那是安全的。

因此，我们使用双星模式的高可用服务器架构拥有两台服务器和跟这两台服务器交流的一组客户端（参见图 5-7）。

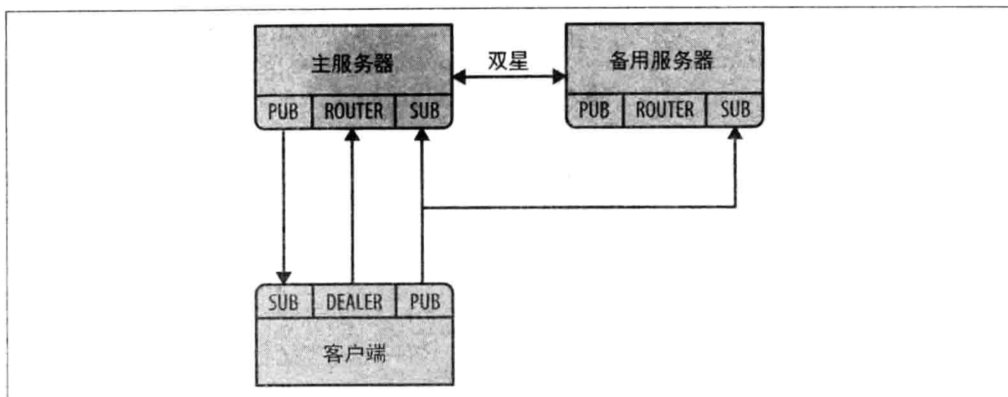


图5-7：高可用的克隆服务器对

示例 5-51 到示例 5-58 说明了克隆服务器的第六个和最后一个模型。

示例5-51：克隆服务器，模型六（clonesrv6.c）

```
//
//  克隆服务器 - 模型六
//

//  构建这个源代码而不创建一个库
#include "bstar.c"
#include "kvmsg.c"
```

在示例 5-52 中，我们定义了一组反应器处理程序和我们的服务器对象结构。

示例5-52：克隆服务器，模型六（clonesrv6.c）：定义

```
//  bstar 反应器处理程序
static int s_snapshots   (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_collector   (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_flush_ttl   (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_send_hugz   (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_new_active  (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_new_passive (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_subscriber  (zloop_t *loop, zmq_pollitem_t *poller, void *args);

//  服务器是由这些属性定义的
typedef struct {
    zctx_t *ctx;                //  上下文包装器
300 zhash_t *kvmap;             //  键 - 值存储区
    bstar_t *bstar;             //  bstar 反应器核
    int64_t sequence;           //  有多少个更新
    int port;                   //  正在上面工作的主端口
```

```
    int peer;                       // 对等节点的主端口
    void *publisher;                // 发布更新和 hugz
    void *collector;                // 收集来自客户端的更新
    void *subscriber;               // 获取来自对等节点的更新
    zlist_t *pending;               // 来自客户端的待定更新
    Bool primary;                   // 如果我们是主服务器，则返回 TRUE
    Bool active;                    // 如果我们是活动服务器，则返回 TRUE
    Bool passive;                   // 如果我们是被动服务器，则返回 TRUE
} clonesrv_t;
```

主任务解析命令行来决定是作为主服务器还是备用服务器启动。为了可靠性，我们正在使用双星模式。这个模式互联两台服务器，使它们能够就哪台是主服务器，哪台是备用服务器达成一致。为了使两台服务器能够在同一台电脑上运行，我们对主服务器和备用服务器使用不同的端口，如示例 5-53 所示。端口 5003/5004 用于互联服务器，端口 5556/5566 用于接收投票事件(在克隆模式中的快照请求),端口 5557/5567 被发布者使用，而端口 5558/5568 被收集器使用。

示例5-53：克隆服务器，模型六（clonesrv6.c）：主任务设置

```
int main (int argc, char *argv [])
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));
    if (argc == 2 && streq (argv [1], "-p")) {
        zclock_log ("I: primary active, waiting for backup (passive)");
        self->bstar = bstar_new (BSTAR_PRIMARY, "tcp://*:5003",
                                 "tcp://localhost:5004");
        bstar_voter (self->bstar, "tcp://*:5556", ZMQ_ROUTER, s_snapshots, self);
        self->port = 5556;
        self->peer = 5566;
        self->primary = TRUE;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        zclock_log ("I: backup passive, waiting for primary (active)");
        self->bstar = bstar_new (BSTAR_BACKUP, "tcp://*:5004",
                                 "tcp://localhost:5003");
        bstar_voter (self->bstar, "tcp://*:5566", ZMQ_ROUTER, s_snapshots, self);
        self->port = 5566;
        self->peer = 5556;
        self->primary = FALSE;
    }
    else {
        printf ("Usage: clonesrv4 { -p | -b }\n");
        free (self);
        exit (0);
```

301

```
    }
    // 主服务器将成为首先活动的
    if (self->primary)
        self->kvmap = zhash_new ();

    self->ctx = zctx_new ();
    self->pending = zlist_new ();
    bstar_set_verbose (self->bstar, TRUE);

    // 设置克隆服务器套接字
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_SUB);
    zsockopt_set_subscribe (self->collector, "");
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    // 设置我们自己的到对等节点的克隆客户端接口
    self->subscriber = zsocket_new (self->ctx, ZMQ_SUB);
    zsockopt_set_subscribe (self->subscriber, "");
    zsocket_connect (self->subscriber, "tcp://localhost:%d", self->peer + 1);
```

建立了我们的套接字之后，先注册双星事件处理程序，然后启动 bstar 反应器。当用户按下 Ctrl-C 或当进程接收到 SIGINT 中断时，就停止它的执行。主任务的主体如示例 5-54 所示。

示例5-54：克隆服务器，模型六（clonesrv6.c）：主任务主体

```
    // 注册状态变化处理程序
    bstar_new_active (self->bstar, s_new_active, self);
    bstar_new_passive (self->bstar, s_new_passive, self);

    // 将另一个处理程序注册到 bstar 反应器
    zmq_pollitem_t poller = { self->collector, 0, ZMQ_POLLIN };
    zloop_poller (bstar_zloop (self->bstar), &poller, s_collector, self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_flush_ttl, self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_send_hugz, self);

    // 启动 bstar 反应器
    bstar_start (self->bstar);

    // 中断，因此关闭
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_destroy (&kvmsg);
    }
    zlist_destroy (&self->pending);
```

```
    bstar_destroy (&self->bstar);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);

    return 0;
}
```
302

```
//  我们处理“ICANHAZ?”请求的过程与 clonesrv5 示例完全相同。
...
```

这个收集器（参见示例 5-55）比在 *clonesrv5* 示例中的更复杂，因为其处理更新的方式取决于服务器是活动的还是被动的。活动服务器立即将更新应用到它的 kvmap，而被动服务器会将它们作为待定的更新进行排队。

示例5-55：克隆服务器，模型六（clonesrv6.c）：收集更新

```
//  如果消息已经在待定清单中，则删除它，然后返回 TRUE;
//  否则返回 FALSE
static int
s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg)
{
    kvmsg_t *held = (kvmsg_t *) zlist_first (self->pending);
    while (held) {
        if (memcmp (kvmsg_uuid (kvmsg),
                    kvmsg_uuid (held), sizeof (uuid_t)) == 0) {
            zlist_remove (self->pending, held);
            return TRUE;
        }
        held = (kvmsg_t *) zlist_next (self->pending);
    }
    return FALSE;
}

static int
s_collector (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (poller->socket);
    if (kvmsg) {
        if (self->active) {
            kvmsg_set_sequence (kvmsg, ++self->sequence);
            kvmsg_send (kvmsg, self->publisher);
```

```
            int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
            if (ttl)
                kvmsg_set_prop (kvmsg, "ttl",
                    "%"PRId64, zclock_time () + ttl * 1000);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: publishing update=%d", (int) self->sequence);
        }
        else {
            //  如果我们已经从活动服务器中得到消息，
            //  则删除它，否则将其保留在待定列表
            if (s_was_pending (self, kvmsg))
                kvmsg_destroy (&kvmsg);
            else
303             zlist_append (self->pending, kvmsg);
        }
    }
    return 0;
}

//  使用与前面 clonesrv5 示例完全相同的代码来清除临时值
...
```

我们每秒给所有订阅者发送一次 HUGZ 消息，以便这些订阅者可以检测我们的服务器是否死机（参见示例 5-56）。如果服务器死机了，那么它们就会切换到备用服务器，而那个服务器将成为活动的。

示例5-56：克隆服务器，模型六（clonesrv6.c）：信号检测

```
static int
s_send_hugz (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_new (self->sequence);
    kvmsg_set_key  (kvmsg, "HUGZ");
    kvmsg_set_body (kvmsg, (byte *) "", 0);
    kvmsg_send     (kvmsg, self->publisher);
    kvmsg_destroy (&kvmsg);

    return 0;
}
```

当从被动切换为活动时，我们要应用待定清单，以便 kvmap 是最新的。当切换到被动时，擦除 kvmap 并从活动进程中抓取一个新的快照，如示例 5-57 所示。

示例5-57：克隆服务器，模型六（clonesrv6.c）：处理状态变化

```
static int
s_new_active (zloop_t *loop, zmq_pollitem_t *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    self->active = TRUE;
    self->passive = FALSE;

    //  停止订阅更新
    zmq_pollitem_t poller = { self->subscriber, 0, ZMQ_POLLIN };
    zloop_poller_end (bstar_zloop (self->bstar), &poller);

    //  将待定清单应用到自己的散列表
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing pending=%d", (int) self->sequence);
    }
    return 0;
}

static int
s_new_passive (zloop_t *loop, zmq_pollitem_t *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zhash_destroy (&self->kvmap);
    self->active = FALSE;
    self->passive = TRUE;

    //  开始订阅更新
    zmq_pollitem_t poller = { self->subscriber, 0, ZMQ_POLLIN };
    zloop_poller (bstar_zloop (self->bstar), &poller, s_subscriber, self);

    return 0;
}
```

304

当得到一个更新时，如果有必要，我们就创建一个新的 kvmap，然后将更新添加到此 kvmap 中（参见示例 5-58）。在这种情况下，我们总是被动的。

示例5-58：克隆服务器，模型六（clonesrv6.c）：订阅者处理程序

```
static int
```

```
s_subscriber (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    //  如果有必要，则获取状态快照
    if (self->kvmap == NULL) {
        self->kvmap = zhash_new ();
        void *snapshot = zsocket_new (self->ctx, ZMQ_DEALER);
        zsocket_connect (snapshot, "tcp://localhost:%d", self->peer);
        zclock_log ("I: asking for snapshot from: tcp://localhost:%d",
                    self->peer);
        zstr_sendm (snapshot, "ICANHAZ?");
        zstr_send (snapshot, ""); // 空白的子树表示获取全部
        while (true) {
            kvmsg_t *kvmsg = kvmsg_recv (snapshot);
            if (!kvmsg)
                break;          //  中断
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                kvmsg_destroy (&kvmsg);
                break;          //  完成
            }
            kvmsg_store (&kvmsg, self->kvmap);
        }
305     zclock_log ("I: received snapshot=%d", (int) self->sequence);
        zsocket_destroy (self->ctx, snapshot);
    }
    //  从待定清单中查找并移除更新
    kvmsg_t *kvmsg = kvmsg_recv (poller->socket);
    if (!kvmsg)
        return 0;

    if (strneq (kvmsg_key (kvmsg), "HUGZ")) {
        if (!s_was_pending (self, kvmsg)) {
            //  如果活动更新比客户端更新早到达，则把它弹到一边，
            //  将活动更新（含序号）存储在待定清单中，
            //  并用它来清除晚到的客户端更新
            zlist_append (self->pending, kvmsg_dup (kvmsg));
        }
        //  如果更新比我们的 kvmap 还新，则应用它
        if (kvmsg_sequence (kvmsg) > self->sequence) {
            self->sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: received update=%d", (int) self->sequence);
        }
        else
```

```
            kvmsg_destroy (&kvmsg);
        }
        else
            kvmsg_destroy (&kvmsg);

    return 0;
}
```

这个模型的源代码只有几百行，但花了相当一段时间才能正常工作。准确地说，构建模型六花了大约一整周的“亲爱的神，这对这本书来说简直太复杂”的钻研。我们几乎已经组建了一切，甚至把厨房收集器都用到了这个小应用程序中。我们拥有故障转移、临时值、子树，等等。让我吃惊的是，前期设计是相当准确的。尽管如此，编写和调试这么多套接字流的细节仍然具有相当大的挑战性。

以反应器为基础的设计从代码中消除了大量繁重的工作，剩下的就是更简单和更容易理解的东西。我们重用了来自第 4 章的 bstar 反应器，整个服务器作为一个线程在运行，因此在运行时没有线程间的怪事，只是一个传递给周围所有处理程序的结构指针（self），它可以开心地做自己的事情。使用反应器的一个很好的副作用是代码，它不太紧密地集成到轮询循环，所以更容易重用。在模式六中，有大块的代码取自模型五。

我将它逐个部件地构建起来，并让每个部件都正常工作后才去构建下一个。因为存在四个或五个主要的套接字流，这意味着需要执行相当多的调试和测试。我只是通过将信息转储到控制台来调试。不要用经典的调试器来逐步调试 ØMQ 应用程序，你需要看到消 306
息流，以明白正在发生的事情。

为了测试，我总是尝试使用 valgrind，它捕获内存泄漏和无效内存访问。在 C 语言中，这是一个大问题，因为不能把它委托给垃圾收集器。使用适当的和一致的抽象，比如 kvmsg 和 CZMQ，将会有极大的帮助。

集群的散列映射协议

虽然服务器模型六几乎就是以前的模型加上双星模式的混搭，但客户端却复杂很多。但是，在我们研究那个任务之前，让我们先来看看最后的协议。我已经在 ZeroMQ RFC 网站将其作为集群的散列映射协议（CHP）（*http://rfc.zeromq.org/spec:12*）的规范编写了。

粗略地说，设计复杂的协议，比如这个协议，有两种方法。第一种方法是将每个流分离成其自己的一组套接字，这就是我们在这里所使用的方法。它的优点是，每个流都是简单和整洁的。缺点是一次性管理多个套接字流可能会相当复杂。使用反应器使得它更简单，但是，这造成了很多必须正确地配合在一起的活动部件。

第二种制作出这样的协议的方法是对一切东西使用单个套接字对。在这个例子中，我已经把 ROUTER 用于服务器，并把 DEALER 用于客户端，然后通过该连接做一切事。它使得一个协议更复杂，但至少其复杂性都在一个地方。在第 7 章，我们将探讨通过一个 ROUTER-DEALER 组合来完成协议的一个例子。

下面让我们来阅读 CHP 规范，本文是从 RFC 直接取得的。需要注意的是，"应该"和"必须"是我们在协议规范中用来指示需求级别的关键字。

目标

CHP 的目的是为跨一个 ØMQ 网络连接的客户端集群提供可靠的发布 - 订阅的基础。它定义了一个由键 - 值对构成的"散列映射"抽象。任何客户端可以随时修改任意键 - 值对，并将更改传播到所有客户端。客户端可以在任何时间加入网络。

架构

CHP 连接一组客户端应用程序和一组服务器。客户端连接到服务器。客户端互相看不到对方。客户端可以随意来去。

端口和连接

服务器必须打开三个端口，如下所示：

- 一个在端口号 P 的 SNAPSHOT 端口（ØMQ ROUTER 套接字）。
307 - 一个在端口号 P + 1 的 PUBLISHER 端口（ØMQ PUB 套接字）。
- 一个在端口号 P + 2 的 COLLECTOR 端口（ØMQ SUB 套接字）。

客户端应该至少打开两个连接：

- 一个到端口号 P 的 SNAPSHOT 连接（ØMQ DEALER 套接字）。
- 一个到端口号 P + 1 的 SUBSCRIBER 连接（ØMQ SUB 套接字）。

客户端可以打开第三个连接，如果它想要更新散列映射：

- 一个到端口号 P + 2 的 PUBLISHER 连接（ØMQ PUB 套接字）。

这个没有在命令中显示的额外的帧将在后面解释。

状态同步

客户端必须通过发送 ICANHAZ 命令到其快照的连接来开始。此命令由如下两帧构成：

ICANHAZ 命令

第 0 帧："ICANHAZ?"
第 1 帧：子树规范

这两帧都是 ØMQ 字符串。子树规范可以是空的。如果不是空的，那么它由一个斜线跟着一个或多个路径段组成，并以一个斜线结束。

服务器必须通过发送零个或多个 KVSYNC 命令到其快照端口来响应 ICANHAZ 命令，后面跟着一个 KTHXBAI 命令。服务器必须使用客户端的身份对每个命令加前缀，如 ØMQ 用 ICANHAZ 命令提供的身份。KVSYNC 命令指定一个键 - 值对，如下所示：

KVSYNC 命令

第 0 帧：键，作为 ØMQ 字符串
第 1 帧：序列号，按照网络顺序的 8 个字节
第 2 帧：<空>
第 3 帧：<空>
第 4 帧：值，作为 blob

序列号没有任何意义，并且可以为零。

KTHXBAI 命令的格式为：

KTHXBAI 命令

第 0 帧："KTHXBAI"
第 1 帧：序列号，按照网络顺序的 8 个字节
第 2 帧：<空>
第 3 帧：<空> 308
第 4 帧：子树规范

该序列号必须是先前发送的 KVSYNC 命令的最大顺序号。

当客户端已经收到了一个 KTHXBAI 命令时，它应该开始从它的订阅者的连接中接收消息，并应用它们。

服务器到客户端的更新

当服务器对它的散列映射更新时，它必须在其发布者套接字上作为 KVPUB 命令广播它。KVPUB 命令有以下形式：

KVPUB 命令

第 0 帧：键，作为 ØMQ 字符串

```
    第 1 帧：序列号，按照网络顺序的 8 个字节
    第 2 帧：UUID，16 个字节
    第 3 帧：属性，ØMQ 字符串
    第 4 帧：值，作为 blob
```

序列号必须严格递增。客户端必须丢弃序列号不严格大于接收到的最后一个 KTHXBAI 或 KVPUB 命令的任何 KVPUB 命令。

UUID 是可选的，第 2 帧可以为空（大小为零）。属性字段的格式为后跟一个换行符的 *name=value* 的零个或多个实例。如果键 - 值对没有属性，则属性字段为空。

如果值为空，客户端应该删除其具有指定键的键 - 值项。

在没有其他更新的时候，服务器应该定期发送 HUGZ 命令，例如，每秒发送一次。HUGZ 命令的格式如下：

```
HUGZ 命令
    -----------------------------------
    第 0 帧："HUGZ"
    第 1 帧：00000000
    第 2 帧：<空>
    第 3 帧：<空>
    第 4 帧：<空>
```

客户端可以把没有 HUGZ 作为该服务器已经崩溃的指示器，参见下面的“可靠性”一小节。

客户端到服务器的更新

当客户端有其散列映射的更新时，它可以作为 KVSET 命令通过其发布者连接将更新发送到服务器。KVSET 命令有以下形式：

309

```
KVSET 命令
    -----------------------------------
    第 0 帧：键，作为 ØMQ 字符串
    第 1 帧：序列号，按照网络顺序的 8 个字节
    第 2 帧：UUID，16 个字节
    第 3 帧：属性，作为 ØMQ 字符串
    第 4 帧：值，作为 blob
```

序列号没有任何意义，并且可以为零。如果使用的是一个可靠的服务器架构，UUID 应该是一个通用唯一标识符。

如果该值为空，那么服务器必须删除其具有指定键的键 - 值项。

服务器应该接受以下属性。

- ttl：指定一个以秒为单位的存活时间。如果 KVSET 命令有一个 `ttl` 属性，服务器应该删除键 - 值对并且广播一个带有空值的 KVPUB，以便当 TTL 已过期时从所有客户端删除此键 - 值对。

可靠性

CHP 可以在如果主服务器出现故障就由备用服务器接管的双服务器配置中使用。CHP 并没有指定用于此故障转移的机制，但是来自第 4 章的双星模式可能会对此有所帮助。

为了有助于服务器的可靠性，客户端可以：

- 在每一个 KVSET 命令中设置一个 UUID。
- 在一段时间内检测 HUGZ 的缺少，并以此作为当前服务器发生故障的指示器。
- 连接到备用服务器，并重新请求一个状态同步。

可扩展性和性能

CHP 被设计为可扩展到大量（数千个）客户端，仅受代理上的系统资源限制。因为所有的更新都通过一台服务器传递，总吞吐量将被限制为在高峰期每秒几百万次的更新，并可能更少。

安全

CHP 不执行任何身份验证、访问控制和加密机制，并且不应在需要这些机制的任何部署中使用。

构建一个多线程栈和 API 310

我们至今使用的客户端栈不够聪明，还不能妥善处理这一协议。当开始执行信号检测时，我们需要一个可以在后台线程中运行的客户端栈。在第 4 章末尾的自由职业者模式中，我们使用了一个多线程的 API，但没有详细解释它。事实证明，当你开始制作更复杂的 ØMQ 协议，如 CHP 时，多线程的 API 是非常有用的。

如果你制作了一个复杂的协议，并且希望应用程序能够正确地实现它，那么大多数开发者在大部分时间都会弄错它。你将会面对很多不快乐的人，他们抱怨你的协议太复杂、太脆弱，并且太难以使用。而如果你给他们一个简单的 API 来调用，你就有机会让他们购买这个 API。

我们的多线程 API 由一个前端对象和一个后台代理组成，它们通过两个 PAIR 套接字来连接（参见图 5-8）。像这样连接两个 PAIR 套接字是非常有用的，所以你的高层次的绑

定也许应该做 CZMQ 所做的工作，这包装了一个“创建带有一个我可以用来发送消息的管道的新线程”的方法。

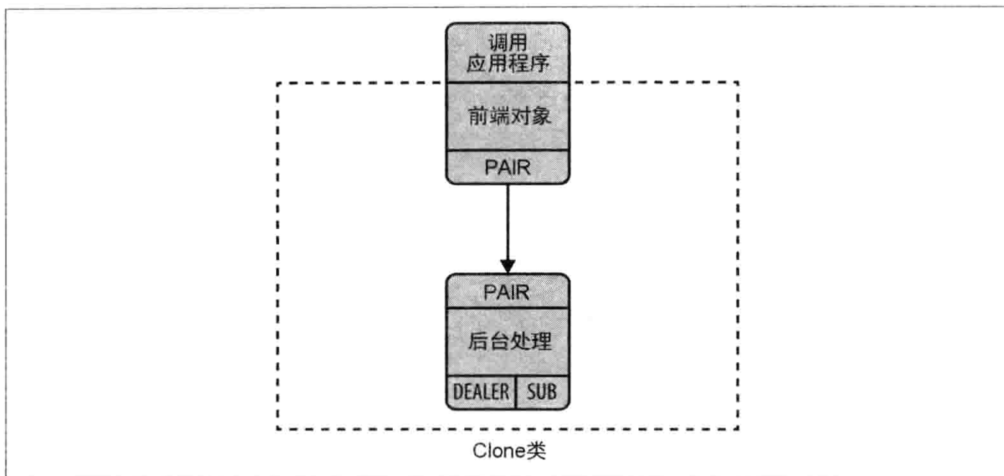


图5-8：多线程的API

在本书中看到的所有多线程 API 都具有相同的形式：

- 对象的构造函数（clone_new()）创建了一个上下文，并启动一个与管道连接的后台线程。它持有管道的一端，所以它可以发送命令到后台线程。
- 后台线程启动一个代理，它实际上是一个从管道套接字，以及任何其他套接字（在这里，是 DEALER 和 SUB 套接字）读取的 zmq_poll() 循环。
- 311 主应用程序线程和后台线程现在只能通过 ØMQ 消息进行通信。按照惯例，前端发送字符串命令，以便类中的每个方法都把它变成发送给后端代理的消息，类似下面这样：

```
void
clone_connect (clone_t *self, char *address, char *service)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, address);
    zmsg_addstr (msg, service);
    zmsg_send (&msg, self->pipe);
}
```

- 如果方法需要一个返回代码，它可以等待来自代理的应答消息。

- 如果代理需要把异步事件发回给前端，我们就在类中添加一个 recv 方法，它等待在前端管道上的消息。
- 我们可能希望公开前端管套接字句柄，以使这个类能被集成到进一步的轮询循环中。否则，任何 recv 方法都会阻塞应用程序。

clone 类与来自第 4 章的 flcliapi 类具有相同的结构，并且添加了来自克隆客户端的最后一个模型的逻辑。如果没有 ØMQ，这种多线程 API 的设计将需要几周非常艰苦的工作。但有了 ØMQ，这只需要用一天或两天的工作量。

clone 类的实际 API 方法是非常简单的：

```
// 创建一个新的克隆类实例
clone_t *
    clone_new (void);

// 销毁一个克隆类实例
void
    clone_destroy (clone_t **self_p);

// 为这个克隆类定义子树（如果有的话）
void
    clone_subtree (clone_t *self, char *subtree);

// 将克隆类连接到一台服务器
void
    clone_connect (clone_t *self, char *address, char *service);

// 在共享散列映射中设置一个值
void
    clone_set (clone_t *self, char *key, char *value, int ttl);

// 从共享的散列映射获取一个值
char *
    clone_get (clone_t *self, char *key);
```
312

示例 5-59 给出了克隆客户端，它现已成为只是一个使用 clone cl 的薄壳。

示例5-59：克隆客户端，模型六（clonecli6.c）

```
// 克隆客户端模型六
#include "clone.c"
#define SUBTREE "/client/"
int main (void)
{
    // 创建分布式散列实例
```

```
    clone_t *clone = clone_new ();
    //  指定配置
    clone_subtree (clone, SUBTREE);
    clone_connect (clone, "tcp://localhost", "5556");
    clone_connect (clone, "tcp://localhost", "5566");
    //  将随机的元组设置到分布式散列中
    while (!zctx_interrupted) {
        //  设置随机值，并检查它是否已被存储
        char key [255];
        char value [10];
        sprintf (key, "%s%d", SUBTREE, randof (10000));
        sprintf (value, "%d", randof (1000000));
        clone_set (clone, key, value, randof (30));
        sleep (1);
    }
    clone_destroy (&clone);
    return 0;
}
```

注意 connect 方法，它指定一个服务器端点。在后台，我们其实是在与三个端口交流。然而，正如 CHP 规定的，三个端口都在连续的端口号上：

- 服务器状态路由器（ROUTER）在端口 P。
- 服务器更新发布者（PUB）在端口 P + 1。
- 服务器更新订阅者（SUB）在端口 P + 2。

因此，我们可以将三个连接折叠到一个逻辑操作中（将其实现为三个独立的 ØMQ 连接调用）。

让我们以用于克隆栈的源代码来结束。这是一段复杂的代码，但当你将它分解成前端对象类和后台代理时，就更容易理解了。前端发送字符串命令（“SUBTREE”、“CONNECT”、
313 “SET”、“GET”）到代理，这个代理处理这些命令并和服务器交流。下面是这个代理的逻辑：

1. 通过从第一台服务器获取快照来启动。
2. 当得到一个快照时，切换到从订阅者套接字读取。
3. 如果没有得到一个快照，那么故障转移到第二个服务器。
4. 在管道和订阅者套接字上轮询。
5. 如果在管道上得到输入，那么就处理来自前端对象的控制消息。
6. 如果在订阅者上得到输入，那么就存储或应用此更新。
7. 如果在一定时间内没有从服务器得到任何东西，那就执行故障转移。
8. 重复直到该进程被 Ctrl-C 中断。

下面是实际的 clone 类的实现。类的结构如示例 5-60 所示。

示例5-60：克隆类（clone.c）

```
/*  =====================================================================
 *  clone - 克隆客户端 API 栈（多线程）
 *  ===================================================================== */

#include "clone.h"

//  如果在这段时间内没有服务器应答，则放弃请求
#define GLOBAL_TIMEOUT  4000    //  毫秒


//  =====================================================================
//  同步的一部分，在我们的应用程序的线程中工作

//  ---------------------------------------------------------------------
//  我们的类的结构

struct _clone_t {
    zctx_t *ctx;                //  上下文包装器
    void *pipe;                 //  通向克隆代理的管道
};

//  这是处理我们真正的克隆类的线程
static void clone_agent (void *args, zctx_t *ctx, void *pipe);
```

示例 5-61 给出了 clone 类的构造函数和析构函数。请注意，我们专门把前端连接到后端代理的管道创建为一个上下文。

示例5-61：克隆类（clone.c）：构造函数和析构函数

```
clone_t *
clone_new (void)
{
    clone_t
        *self;

    self = (clone_t *) zmalloc (sizeof (clone_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, clone_agent, NULL);
    return self;
}

void
clone_destroy (clone_t **self_p)
```

314

```
{
    assert (self_p);
    if (*self_p) {
        clone_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}
```

示例 5-62 中的 subtree 方法指定用于快照和更新的一个子树，这是当子树规范被作为第一个命令发送到服务器时，我们必须在连接到服务器之前做的。此方法发送一个 [SUBTREE][Subtree] 命令给代理。

示例5-62：克隆类（clone.c）：子树方法

```
void clone_subtree (clone_t *self, char *subtree)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SUBTREE");
    zmsg_addstr (msg, subtree);
    zmsg_send (&msg, self->pipe);
}
```

connect 方法(参见示例 5-63)连接到新的服务器端点。我们最多可以连接到两台服务器。此方法发送一个 [CONNECT][endpoint][service] 命令到代理。

示例5-63：克隆类（clone.c）：连接方法

```
void
clone_connect (clone_t *self, char *address, char *service)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, address);
    zmsg_addstr (msg, service);
315 zmsg_send (&msg, self->pipe);
}
```

set 方法（参见示例 5-64）设置在共享散列映射中的新值。它通过给代理发送一个 [SET][key][value][ttl] 命令，由代理来完成实际的工作。

示例5-64：克隆类（clone.c）：设置方法

```
void
clone_set (clone_t *self, char *key, char *value, int ttl)
```

```
{
    char ttlstr [10];
    sprintf (ttlstr, "%d", ttl);

    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SET");
    zmsg_addstr (msg, key);
    zmsg_addstr (msg, value);
    zmsg_addstr (msg, ttlstr);
    zmsg_send (&msg, self->pipe);
}
```

get 方法（参见示例 5-65）在分布式散列表中查找一个值。它给代理发送一个 [GET][key] 命令并等待响应值。如果没有可用的值，这个方法最终将返回 NULL。

示例5-65：克隆类（clone.c）：get方法

```
char *
clone_get (clone_t *self, char *key)
{
    assert (self);
    assert (key);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "GET");
    zmsg_addstr (msg, key);
    zmsg_send (&msg, self->pipe);

    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *value = zmsg_popstr (reply);
        zmsg_destroy (&reply);
        return value;
    }
    return NULL;
}
```

后端代理管理一组服务器，这是我们使用简单类模型实现的，如示例 5-66 所示。

示例5-66：克隆类（clone.c）：与服务器配合工作 316

```
typedef struct {
    char *address;              //  服务器地址
    int port;                   //  服务器端口
    void *snapshot;             //  快照套接字
    void *subscriber;           //  传入的更新
    uint64_t expiry;            //  服务器过期时间
```

```
    uint requests;              //  发出的快照请求数量
} server_t;

static server_t *
server_new (zctx_t *ctx, char *address, int port, char *subtree)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));

    zclock_log ("I: adding server %s:%d...", address, port);
    self->address = strdup (address);
    self->port = port;

    self->snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (self->snapshot, "%s:%d", address, port);
    self->subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "%s:%d", address, port + 1);
    zsockopt_set_subscribe (self->subscriber, subtree);
    return self;
}

static void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->address);
        free (self);
        *self_p = NULL;
    }
}
```

示例 5-67 显示了后端代理本身的实现。

示例5-67：克隆类（clone.c）：后端代理类

```
//  我们将与之交流的服务器数量
#define SERVER_MAX       2

//  如果服务器沉默了这么长时间，它就被认为是死机了
#define SERVER_TTL       5000    //  毫秒

//  我们可以处于的状态
#define STATE_INITIAL        0   //  在向服务器请求状态前
#define STATE_SYNCING        1   //  从服务器获取状态
317 #define STATE_ACTIVE         2   //  从服务器获取新的更新
```

```
typedef struct {
    zctx_t *ctx;                // 上下文包装器
    void *pipe;                 // 回到应用程序的管道
    zhash_t *kvmap;             // 实际键 / 值表
    char *subtree;              // 子树规范，如果有的话
    server_t *server [SERVER_MAX];
    uint nbr_servers;           // 0 至 SERVER_MAX
    uint state;                 // 当前状态
    uint cur_server;            // 是否是活动的，服务器 0 或 1
    int64_t sequence;           // 被处理的最后一个 kvmsg
    void *publisher;            // 发出的更新
} agent_t;

static agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->kvmap = zhash_new ();
    self->subtree = strdup ("");
    self->state = STATE_INITIAL;
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    return self;
}

static void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        int server_nbr;
        for (server_nbr = 0; server_nbr < self->nbr_servers; server_nbr++)
            server_destroy (&self->server [server_nbr]);
        zhash_destroy (&self->kvmap);
        free (self->subtree);
        free (self);
        *self_p = NULL;
    }
}
```

示例 5-68 中的代码处理来自前端的不同控制信息——SUBTREE、CONNECT、SET 和 GET：

示例5-68：克隆类（clone.c）：处理控制消息

```
static int
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);
    if (command == NULL)
318     return -1;      //  中断

    if (streq (command, "SUBTREE")) {
        free (self->subtree);
        self->subtree = zmsg_popstr (msg);
    }
    else
    if (streq (command, "CONNECT")) {
        char *address = zmsg_popstr (msg);
        char *service = zmsg_popstr (msg);
        if (self->nbr_servers < SERVER_MAX) {
            self->server [self->nbr_servers++] = server_new (
                self->ctx, address, atoi (service), self->subtree);
            //  将更新广播给所有已知服务器
            zsocket_connect (self->publisher, "%s:%d",
                address, atoi (service) + 2);
        }
        else
            zclock_log ("E: too many servers (max. %d)", SERVER_MAX);
        free (address);
        free (service);
    }
    else
```

当设置一个属性时，我们将新的键 - 值对推送到所有连接的服务器，如示例 5-69 所示。

示例5-69：克隆类（clone.c）：设置和获取命令

```
        char *key = zmsg_popstr (msg);
        char *value = zmsg_popstr (msg);
        char *ttl = zmsg_popstr (msg);
        zhash_update (self->kvmap, key, (byte *) value);
        zhash_freefn (self->kvmap, key, free);

        //  将键 - 值对发送给服务器
        kvmsg_t *kvmsg = kvmsg_new (0);
        kvmsg_set_key  (kvmsg, key);
        kvmsg_set_uuid (kvmsg);
        kvmsg_fmt_body (kvmsg, "%s", value);
```

```
        kvmsg_set_prop (kvmsg, "ttl", ttl);
        kvmsg_send     (kvmsg, self->publisher);
        kvmsg_destroy (&kvmsg);
        free (ttl);
        free (key);             //  值由散列表拥有
    }
    else
    if (streq (command, "GET")) {
        char *key = zmsg_popstr (msg);
        char *value = zhash_lookup (self->kvmap, key);
        if (value)
            zstr_send (self->pipe, value);
        else
            zstr_send (self->pipe, "");
        free (key);
        free (value);
    }
    free (command);
    zmsg_destroy (&msg);
    return 0;
}
```

319

异步代理（参见示例 5-70）管理一个服务器池，并在应用程序请求它时，处理请求 - 应答对话。

示例5-70：克隆类（clone.c）：后端代理

```
static void
clone_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    while (true) {
        zmq_pollitem_t poll_set [] = {
            { pipe, 0, ZMQ_POLLIN, 0 },
            { 0,    0, ZMQ_POLLIN, 0 }
        };
        int poll_timer = -1;
        int poll_size = 2;
        server_t *server = self->server [self->cur_server];
        switch (self->state) {
            case STATE_INITIAL:
                //  在这个状态下，如果我们要与一台服务器交流，
                //  我们就向服务器请求一个快照……
                if (self->nbr_servers > 0) {
                    zclock_log ("I: waiting for server at %s:%d...",
```

```
                    server->address, server->port);
                if (server->requests < 2) {
                    zstr_sendm (server->snapshot, "ICANHAZ?");
                    zstr_send  (server->snapshot, self->subtree);
                    server->requests++;
                }
                server->expiry = zclock_time () + SERVER_TTL;
                self->state = STATE_SYNCING;
                poll_set [1].socket = server->snapshot;
            }
            else
                poll_size = 1;
            break;

        case STATE_SYNCING:
            //  在这个状态下，我们读取快照并且期望服务器响应，否则执行故障转移
            poll_set [1].socket = server->snapshot;
            break;

        case STATE_ACTIVE:
            //  在这个状态下，我们从订阅者读取并且期望服务器给出 hugz，
            //  否则执行故障转移
            poll_set [1].socket = server->subscriber;
            break;
    }
    if (server) {
        poll_timer = (server->expiry - zclock_time ())
                   * ZMQ_POLL_MSEC;
        if (poll_timer < 0)
            poll_timer = 0;
    }
```

320

现在，我们准备处理传入的消息，如示例 5-71 所示。如果在超时时间内没有收到来自于服务器的任何东西，这意味着服务器已经死机了。

示例5-71：克隆类（clone.c）：客户端轮询循环

```
    int rc = zmq_poll (poll_set, poll_size, poll_timer);
    if (rc == -1)
        break;              //  上下文已被关闭

    if (poll_set [0].revents & ZMQ_POLLIN) {
        if (agent_control_message (self))
            break;          //  中断
    }
```

```
        else
        if (poll_set [1].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (poll_set [1].socket);
            if (!kvmsg)
                break;          //  中断

            //  来自服务器的任何东西重置其过期时间
            server->expiry = zclock_time () + SERVER_TTL;
            if (self->state == STATE_SYNCING) {
                //  在快照中存储，直到我们已完成处理
                server->requests = 0;
                if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                    self->sequence = kvmsg_sequence (kvmsg);
                    self->state = STATE_ACTIVE;
                    zclock_log ("I: received from %s:%d snapshot=%d",
                        server->address, server->port,
                        (int) self->sequence);
                    kvmsg_destroy (&kvmsg);
                }
                else
                    kvmsg_store (&kvmsg, self->kvmap);
            }
            else
            if (self->state == STATE_ACTIVE) {
                //  丢弃序号不对的更新，包括 hugz
                if (kvmsg_sequence (kvmsg) > self->sequence) {
                    self->sequence = kvmsg_sequence (kvmsg);
                    kvmsg_store (&kvmsg, self->kvmap);
                    zclock_log ("I: received from %s:%d update=%d",
                        server->address, server->port,
                        (int) self->sequence);
                }
                else
                    kvmsg_destroy (&kvmsg);
            }
        }
        else {
            //  服务器已死机，故障转移到下一台服务器
            zclock_log ("I: server at %s:%d didn't give hugz",
                server->address, server->port);
            self->cur_server = (self->cur_server + 1) % self->nbr_servers;
            self->state = STATE_INITIAL;
        }
    }
    agent_destroy (&self);
}
```

321

第2部分 <323>

使用ØMQ的软件工程

本书的第2部分研究使用ØMQ的软件工程。我将为大家介绍一套软件开发技术，并用能够工作的实例展示它们，从ØMQ本身开始，以一个分布式应用程序通用框架结束。这些技术的许可证都是独立的，虽然开源放大了它们。

第 6 章 325

ØMQ社区

人们有时会问我 ØMQ 有什么特别之处。我的标准答案是确切的：对于我们必须解决的棘手问题，“我们如何制作满足 21 世纪需求的分布式软件呢？”ØMQ 可以说是最好的解决方案。但更重要的是，ØMQ 是因为它的社区而特殊。这是最终将狼群与绵羊分开的东西。

主要的开源开发模式有三种。第一种模式是大企业贡献代码为别人来打破市场格局。这是 Apache 基金会的模式。第二种模式是微型的团队或小公司建立自己的梦想。这是最常见的开源模式，它可以被非常成功地商业化。最后一种模式是云集在一个问题领域上的积极和多样的社区。这是 Linux 的模型，也是我们渴望 ØMQ 拥有的模型。

怎么强调工作中的开源社区的力量和持久性都不为过。真的似乎并不存在比这更好的制作长期发展的软件的方法。社区不仅选择最好的问题来解决，而且它在最低限度上仔细地解决了它们，然后在之后几年，甚至几十年中负责维护这些问题的答案，直到它们不再重要。然后，它默默地把它们放走。

为了真正从 ØMQ 受益，你需要了解社区。在使用它的过程中，某些时候，你会希望提交一个补丁程序、问题或插件。你可能要寻求他人的帮助。你可能会想将你的业务的一部分寄托在 ØMQ 上，而当我告诉你，这个社区比产品背后的公司更为重要时（尽管我是那家公司的首席执行官，但还是会这么说），这一点应该是明显的。

在本章中，我将从多个角度去观察我们的社区。我将通过对我们称之为“C4”（*http://rfc.zeromq.org/spec:16*）的协作合同进行详细解释来做总结。你应该发现这个讨论对自己的工作非常有用。我们还成功地为闭源项目改编了 ØMQ C4 过程。

326 我们将讨论：

- 作为一组项目的 ØMQ 的粗略结构
- 什么是“软件架构”的真正含义
- 为什么使用 LGPL 许可证，而不是 BSD 许可证
- 我们是如何设计并培育 ØMQ 社区的
- 在 ØMQ 背后的业务
- 谁拥有 ØMQ 源代码
- 如何为 ØMQ 制作和提交一个补丁程序
- 谁控制什么补丁程序真实地被加入 ØMQ
- 如何保证与旧代码的兼容性
- 为什么我们不使用公共的 Git 分支
- 谁决定 ØMQ 的路线图
- 一个对 `libzmq` 进行更改的实际示例

ØMQ 社区的架构

你知道 ØMQ 是 LGPL 许可项目。事实上，它是围绕 `libzmq` 这个核心库构建的项目的集合。我会把这些项目想象为扩展的星系：

- 处于核心的是 `libzmq`。它是用 C++ 编写的，带有一个低级别的 C API。该代码是令人讨厌的，主要是因为它是高度优化的，也因为它是用 C++ 这种本身含蓄而难懂的语言编写的。Martin Sustrik 最初为它编写了大量的代码。今天，有数十人负责维护它的不同部分。
- 围绕 `libzmq` 大约有 50 个*绑定程序*。这些程序都是为 ØMQ 创建更高级别的 API，或者至少是将低级别的 API 映射成其他语言的个别项目。绑定程序的质量差别较大，从实验性的到完全优秀的。也许令人印象最深刻的绑定程序是 PyZMQ（*http://github.com/zeromq/pyzmq*），这是建立在 ØMQ 上面的首批社区项目之一。如果你是一个绑定程序的编写者，你真的应该学习 PyZMQ 并渴望使你的代码和社区尽可能出色。
- 很多语言都有多个绑定程序（至少是 Erlang、Ruby、C#），随着时间的推移，它们由不同的人编写或采取不同的方法。我们不以任何方式管控这些，不存在“官方”的绑定程序。你可以通过使用一个或另一个，促进它，或忽略它来投票。
- 有一系列 `libzmq` 的重新实现，从 JeroMQ 开始，这是一个完整的 Java 翻译的库，它
327 现在是 NetMQ 这个 C# 堆栈的基础。这些原生栈提供与 `libzmq` 相似或相同的 API，并采用相同的协议（ZMTP）来交流。

- 在绑定程序的上部有很多使用ØMQ或在它基础上建立的项目。要了解在某些方面使用ØMQ的项目和原型项目，请参见wiki上“Labs”页面中的一个长长的清单。在那里有框架，类似Mongrel2的Web服务器，类似Majordomo的代理，和类似Storm的企业开源工具。

`libzmq`、大部分的绑定程序，以及一些外部项目都托管在GitHub上的ØMQ社区“组织”（*https://github.com/organizations/zeromq*）中。这个组织是由一群最高级的绑定作者组成的小组“运营”的。那里的运营工作量实际上是非常少的，因为它几乎是完全自我管理的并且这些天都没有冲突。

我创办的iMatix公司，它在社区中扮演特定的角色。我们拥有注册商标，并谨慎地执行它们，以确保，如果你下载了一个自称“ZeroMQ”的包，就可以信任你得到的东西。在罕见的场合，人们也许认为“自由软件”是指没有财产受到威胁，也没有人愿意去捍卫它的软件。但是，从这一章中你会明白的一件事是我们有多么认真地对待我们的软件背后的过程（我说的“我们”的意思是指一个社区，而不是一家公司）。iMatix通过对任何自称为“ZeroMQ”或“ØMQ”的东西强制执行这个过程来支持社区。我们也把时间和金钱用到软件和包装上，至于原因我稍后会解释。

这不是一个慈善工作。ØMQ是一个以赢利为目的的项目，并且是一个非常有利可图的项目。利润广泛分布在所有那些在这方面投资的人中间。它就是这么简单：只要花时间去成为ØMQ专家，或在ØMQ上面建立一些有用的东西，你就会发现你作为个人，或团队，或公司日益增长的价值。iMatix与在社区中的所有其他人享有相同的好处。这对于每个人，都是一个双赢的结果，而我们的竞争对手除外，他们会发现自己面对的是他们不能击败并且无法真正逃脱的威胁。ØMQ将主宰未来的大型分布式软件的世界。

我的公司不只是拥有社区的支持，我们还建立了社区，这是有意为之的工作。在2007年的原始ØMQ白皮书中有两个项目。第一个项目是技术性的，负责制造一个更好的消息传递系统。第二个项目是建立一个社区，使它可以带领软件走向占主导地位的成功。软件会消亡，但社区会继续生存。

如何制作真正的大型架构

有人说（至少由大声读这句话的人说出），存在两种制作真正的大型软件的方法。第一种方法，是在聪明人的帝国中抛出大量的金钱和问题，并希望脱颖而出的不是另一个职场杀手。如果你在大量经验的基础上构建，也保持了你的团队的团结，并且目标不是技术上的辉煌，并且还非常幸运，那么这个方法能够正常工作。

但用别人亿万的金钱来赌博并不适合每个人。对于我们这些想要构建大型软件的人，还 328

有第二种方法，这就是开源，更具体地说，是自由软件。如果你询问软件许可证的选择是如何与你构建的软件的规模相关的，那么这是个正确的问题。

杰出和富有远见的 Eben Moglen 曾经说过，大致意思是，一个自由软件许可证是在其上建立社区的合同。大约 10 年前，当我听到这句话时，也产生了以下想法：我们可以特意培育自由软件社区吗？

十年后，得到的答案是“是”，而且这几乎成了一门科学。我说“几乎”是因为我们还没有足够的证据来表明人们特意采用一个在文档中记录的，可重复的过程这样做。这是我想要在社会结构（*http://softwareandsilicon.com/chapter:2#toc5*）（译者注：本书作者的另一本书）中做的。ØMQ 源自 Wikidot，源自数字标准组织（Digistan），也源自自由信息基础设施基金会（又名 FFII，一个为反对软件专利而斗争的非政府组织）。这些都源自许多类似 Xitami 和 Libero 的不太成功的社区项目。从漫长的职业生涯中，我得到的主要启发是：如果你想建立真正的大规模和持久的软件，那就把目标放在建立一个自由软件社区上。

软件架构的心理学

Dirkjan Ochtman 给我指出了维基百科上软件架构的定义（*http://en.wikipedia.org/wiki/Software_architecture*）：“一组需要根据系统推理的结构，其中包括软件元素，它们之间的关系，以及两者的性能。”对于我来说，这索然无味和循环的行话本身就是一个很好的例子，它说明我们对于究竟是什么造就了成功的大型软件架构的了解少得有多么可怜。

建筑学是制作大型人工构筑物供人类使用的艺术和科学。如果在 30 年制作越来越大的软件系统的工作中，我有一件学到并成功应用的事，那就是：软件是关于人的。大型构筑物本身是没有意义的，它们如何对人类利用发挥作用才重要。而在软件方面，人类利用开始于制作软件本身的程序员。

软件架构的核心问题是由人的心理，而不是技术驱动的。我们的心理影响工作的方式有很多。我可以指出的一种影响方式是：当团队变得更大或当他们必须跨越更大的地域工作时，他们似乎变得更愚蠢。这是否意味着团队越小就越有效呢？那么像 ØMQ 那样一个庞大的全球社区又是怎么成功运作的呢？

ØMQ 社区不是偶然产生的。它是一个深思熟虑的设计，是我在布拉迪斯拉发的一个地窖中，在代码面世的初期所做的贡献。这个设计是基于我的“社会结构”宠物科学，它在维基百科中定义为“鼓励促进一些目标或一组目标的所需范围内的社会行为的有意识
329 的环境设计”。通过类比传统架构，我的定义是：“用于规划、设计和培育在线社区的过程和产品”。

社会结构的原则之一是，我们如何组织比我们是谁更重要。同样的团队，通过不同方式来组织，可以产生完全不同的结果。我们就像在 ØMQ 网络中的节点，而我们的沟通模式会对我们的业绩产生重大影响。普通人如果连接良好，其表现可以远远超越用差的模式连接的一组专家。如果你是一个较大的 ØMQ 应用程序的架构师，你将不得不去帮助别人找到合适的合作模式。做对了这件事，你的项目就能够成功。做错了这件事，你的项目就会失败。

两个最重要的心理因素是我们人类真的很不善于理解复杂性，并且我们真的善于用合作来对大问题分而治之。我们是高度社会化的类人猿，有点聪明，但只在合适的人群中。

所以，下面是我的软件架构的心理元素的简短列表。

愚蠢

我们的智力的带宽是有限的，所以我们都在某些方面无知。由此可见，架构必须简单易懂。每次设计架构都要遵循简单胜过功能这一头号法则。如果你不能在冷灰的星期一早上喝咖啡前弄明白某个架构，那么就表明它实在是太复杂了。

自私

我们的行为只是出于自身利益，因此架构必须为有利于整体利益的自私行为创造空间和机会。自私往往是间接而含蓄的。例如，我会花时间帮助别人来理解某样东西，因为由此换来的收获可能抵得上我以后几天时间的工作。

懒惰

我们做了很多假设，其中有许多是错误的。当我们可以花最少的努力获得结果或快速测试假设的时候，我们是最幸福的，因此架构必须使这成为可能。具体而言，这意味着它必须是简单的。

嫉妒

我们嫉妒别人，这意味着我们将克服我们的愚蠢和懒惰来证明别人错了，并在竞争中击败他们。因而，架构必须基于任何人都可以理解的公平规则来创造公开竞争的空间。

恐惧

我们不愿意承担风险，特别是如果它们可能会让我们看起来愚蠢。对失败的恐惧是人们整体愚蠢地服从并遵循一个团队的一个主要原因。架构应该使得安静的实验方便又便宜，让人们有取得成功的机会，而不会因失败而受惩罚。

互惠

我们将在艰苦的工作，甚至金钱方面做出额外的付出以惩罚作弊并执行公平的规则。 330

架构应在很大程度上有章可循，它告诉人们如何一起工作，而不是做什么工作。

服从

出于恐惧和懒惰，我们去服从是最幸福的，这意味着如果该模式是很好的，有清楚的解释和记录，以及公平地强制执行，我们每次都会自然地选择正确的道路。

骄傲

我们强烈地意识到我们的社会地位，我们将努力避免在公众面前显得愚蠢或不称职。架构必须确保我们制作的每一个部件都有我们的名字，所以我们就会对别人会怎么说我们的工作紧张得夜不能寐。

贪婪

我们最终是经济的动物（见自私），所以架构必须给我们投资于促成这件事情的经济激励。也许它提高了我们作为专家的声誉，也许它是通过贡献一些技巧或组件来实际上赚钱。激励具体是什么不要紧，但一定要有经济诱因。请把架构视为一个市场，而不是一个工程设计。

在组织或团队内，这些策略不仅在大范围，而且在小范围中起作用。

合同

这是一个故事。它发生在我的同事的一个朋友的表哥的大姐夫身上。他的名字过去是，现在仍然是，帕特里克。

帕特里克是一名计算机科学家，他拥有高级网络拓扑方面的博士学位。他花了两年时间用自己的积蓄构建了一个新产品，并选择了BSD许可证，因为他相信这会让他的产品得到更广泛的采用。他在他的阁楼中工作，个人花费了巨大的精力和财力，并自豪地发布了他的工作成果。人们对此鼓掌，因为这是真正的梦幻般的产品，他的邮件列表很快就充斥着活动、补丁程序和快乐的喋喋不休的讨论。许多公司告诉他，他们是如何用他的工作成果来节省数百万美元的。他们中的有些人甚至付费请他来做咨询和培训。他应邀在会上发言，并开始收集刻有他名字的徽章。他开始做小生意，聘请了一位朋友和他一起工作，并梦想将它做大做强。

后来有一天，有人向他介绍了一个新项目，它是GPL许可的，这个项目派生了他的工作成果，并是在他的产品之上改善的。他于是恼怒和不安，并责问人们——同样是开放资源的家伙！——怎么能这么无耻地窃取他的代码。有关于重新把BSD代码许可为GPL
331 代码甚至是否合法的问题存在长期争论。结果是，这么做是合法的。他试图忽略该新项目，但随后他很快意识到，来自该项目的新补丁程序甚至不能合并回他的产品中！

更糟的是，那个 GPL 的项目得到了普及，而他的一些核心贡献者为该项目制作了第一个小的，然后是更大的修补软件。他仍然不能利用那些变化，他感到自己被遗弃了。帕特里克陷入了低潮，他的女友因为一个古怪的名叫帕特里斯的国际货币交易商而离开了他，他停止了该项目的所有工作。他觉得遭到了背叛，整个人彻底苦不堪言。他解雇了他的朋友，这位朋友觉得这相当不好。最后，帕特里克在一家云公司得到了一份项目经理的工作，而在 40 岁的时候，他已经停止了编程，甚至都失去了编程这方面的乐趣。

可怜的帕特里克，我为他感到惋惜。然后我问他："你为什么不选择 GPL 呢？"他回答说："因为这是一个严格的病毒许可证。"我告诉他，"虽然你能拥有一个博士学位，你也许是我的同事朋友的表哥的大姐夫，但你是一个白痴，并且莫尼克离开你是聪明的。你发布你的工作成果并邀请人们来快乐地窃取你的代码，只要他们在所生产的产品中保持"请偷走我的代码"这类句子，可当人们的确这么做了的时候，你却生气了。更糟的是，你是一个伪君子，因为当他们偷摸这样做时，你很开心，但是当他们公开这么做时，你却觉得遭到背叛了。"

看到你的辛勤工作被一个聪明的团队捕获，然后用它来对付你，这是极其痛苦的，那为什么还要使这成为可能呢？每一个使用 BSD 代码的专有项目都会捕获它。公共 GPL 派生也许是更丢脸的，但现在的结果完全是自找的。

BSD 好比食物，它实际上在低声私语"来吃我吧"！想象一块奶酪方砖，当它位于一个世界上最好的啤酒的空瓶旁边时可能会这么做，这啤酒当然是 Orval，它是由一个古老而几乎绝迹的，名叫 Les Gars La-Bas Qui Brassents l'Orval 的沉默比利时和尚酿造的。BSD 许可，正如它的近似克隆 MIT/X11，是由一所没有利润动机的大学（伯克利）专门设计的，它泄露工作成果和努力。这是一种低于其成本价来推广被补贴的技术的方式，希望通过对被低估的代码的倾销来帮助别人打破市场格局。BSD 是一种优秀的战略工具，但只有当你是一个资金雄厚的大机构时，才能够采用这一种方案。Apache 许可是穿马甲的 BSD。

对于我们这样的小企业，把我们的投资看得像子弹一样宝贵，泄露工作成果和努力是不能接受的。打破市场格局虽然伟大，但我们不能补贴我们的竞争对手。BSD 网络栈最终会将 Windows 放在互联网上。我们无法承受与那些应该自然成为我们盟友的人战斗。我们不能让基本的业务发生错误，因为，这最终意味着我们不得不解雇别人。

这都归结为行为经济学和博弈论。我们选择的许可修改了使用了我们工作成果的那些人 332
的经济性。在软件行业有朋友、敌人和食物。BSD 让大多数人把我们看作午餐。封闭源代码让大多数人把我们看作敌人（你喜欢为软件付给人钱吗？）。但是，GPL 让世界上大多数人成为我们的盟友，唯一的例外是帕特里克。ØMQ 的任何派生的许可都与 ØMQ

兼容，到了我们鼓励把派生作为实验的宝贵工具的地步。是的，看到有人试图带球跑出去，这可能是怪异的，但这里的秘密在于：我可以在任何时候拿回我想要的东西。

过程

如果你已经接受了我迄今为止的论点，那么太棒了！现在，我将解释我们实际建立一个开放源码社区的粗略过程。这就是我们如何把 ØMQ 社区建设（或培育，或轻轻地操纵）成为现实的过程。

作为一个社区的领导者，你的目标是激励人们走出去并探索，以确保他们能够安全地这样做，并且不会干扰他人，在他们做出成功的发现时奖励他们，并确保他们与其他人分享他们的知识（而不是因为我们要求他们，也不是因为他们大方，而是因为这是规定）。

这是一个迭代过程。你制作了一个小产品，花的是你自己的成本，但将它放在公众视野中。然后你围绕该产品建立一个小社区。如果你获得了一个小的，但真正的成功，然后社区就可以帮助你设计和构建其下一个版本，并且社区也会成长得更大。然后社区再帮你构建下下一个版本，依此类推。这是显而易见的，你仍然是社区的一部分，可能甚至是最大的贡献者，但你尝试对物质成果施加的控制越多，想参加社区的人就会越少。请在有人决定你是他们要解决的下一个麻烦问题之前规划好自己的退休。

疯狂，美丽，并且容易

你需要一个足够疯狂和简单的目标，足以让人们在早上从床上爬起来。你的社区必须吸引最优秀的人才，而且需要一些特别的东西。对于 ØMQ，我们曾说过我们的目标是做出“有史以来最快的消息传递”软件，它有资格作为一个很好的激励。如果我们表示我们想要做出“廉价和灵活地连接你整个企业中的移动部件的智能传输层”，那我们就失败了。

那么你的工作一定要漂亮，马上有用，并且有吸引力。你的贡献者是那些只想稍微超出他们现状来做探索的用户。务必使之简单、优雅和极其干净。人们运行或使用你的工作成果的经历应该是一个情绪化的经历。他们应该感到一些东西，如果你已经正确地解决了甚至只是一个大问题，而这个问题是直到那时他们还没有完全意识到是他们所面临的，你就会拥有他们灵魂的一小部分。

333 它也必须易于理解、使用和加入。太多的项目有接入障碍：请将心比心，并领会他们来到你的网站的全部原因，心想：“嗯，有趣的项目，但是 ......”然后离开。你希望他们留下来，并尝试它，哪怕只尝试一次。请使用 GitHub，并把议题跟踪器放在那里。

如果你把这些事情都做好了，你的社区将是聪明的，但更重要的是，它将包括在智力上

和地域上不同的人群。这非常重要，因为一群志同道合的专家不能很好地探索问题的领域，他们往往会犯大错误。而多样性在任何时间都胜过教育。

陌生人，遇见陌生人

两个人共同为某个事情工作需要多少前期协议呢？在大多数组织中，这需要很多。但是你可以把这个成本降低到接近零，然后人们就可以开展合作，这种合作不需要见面，不用发起一个电话会议，也不用会议或出差讨论角色和职责。

你需要由像我这样的玩世不恭的人来设计一个写得很好的规则来迫使陌生人开展互利合作，而不是产生冲突。GPL 是一个良好的开端。GitHub 及其派生 / 合并策略是一个很好的跟进。然后你会想要类似我们的 C4 规则手册的东西，以控制如何实际开展工作。

C4（我现在将它用于每一个新的开源项目）中有很多人们犯的常见错误的详细的和验证过的答案。透明度对于获得信任是必不可少的，而信任对于获得扩展是必不可少的。通过迫使每一个变化都通过一个单一的透明过程做出，你就在结果中建立了真正的信任。

许多开源开发者犯的另一大错误是将自己置于他人之上。“我创办这个项目，因此我的智力优于他人。”这不只是无耻和粗鲁的，并且通常是不准确的，它也是坏事。规则必须没有差别地同样适用于每个人。你是社区的一部分。作为一个项目的创始人，你的工作不是将你对产品的愿景强加给别人，而是要确保规则是好的，诚实的，并使之强制执行。

无限的财富

知识产业中最悲伤的一个神话是，创意是财产的一个智力形式。这是本来应该随着奴隶制一起丢弃的中世纪的废话，但遗憾的是，它仍然使太多厉害的人挣到太多的钱。

创意不值钱。确实，作为财富的智力工作是我们在建设一个市场时做的辛勤工作。“你杀了什么你就吃什么”是鼓励人们努力工作的正确模式。无论是在项目得到的道德权威，
从咨询中挣的钱，或者将商标卖给一些富有的大公司：如果你制作了它，你就拥有它。 334
但你真正拥有的是你参加你的项目的“脚步声”，这最终定义你的力量。

要做到这一点，需要无限的自由空间。值得庆幸的是，GitHub 为我们解决了这个问题，为此，我至死都要感恩（在生活中有很多理由要感恩，我不会在这里列出它们，那是因为这本书只剩一百多页，但这个是必须列出的）。

你可以对每个项目都有较少的所有者的很多小项目的集合进行扩展，但你不能同样地扩展有许多所有者的单个项目。当我们拥抱派生时，一个人只需一次点击就可以成为“所有者”。现在，他们只需要通过展示其独特的价值说服别人加入即可。

所以，在 ØMQ 中，我们的目标是可以很容易地编写在核心库之上的绑定，我们不再试图自己制作这些绑定。这为制作绑定的其他人创造了空间，让他们成为这些绑定的所有者，并获得荣誉。

照管和培育

我希望社区可以百分之百地自动掌握方向，也许有一天这会实现，但目前情况并非如此。对 ØMQ 而言，我们非常接近这个目标，但是从我的经验来看，社区需要四种照管和培育。

- 首先，仅仅是因为大多数人都太友善了，所以我们需要某种象征性的领导或所有者来在发生冲突的时候提供最终裁决。通常是社区的创始人承担这个角色。我已经看到这对自选举的一组“长老”有效，但老男人都喜欢说很多话。我见过社区在谁负责的问题上发生分裂，而建立董事会法人和类似这样的机构似乎使争论更糟地失控，而不是改善这种状况，也许是因为似乎有更多仗要打。一个自由软件的真正好处是，它总是可重新结合的，所以人们不用争夺一个馅饼，而只要派生这个馅饼即可。
- 其次，社区需要生存的规则，因此，他们需要一个律师来制定这些并将其写下来。规则是至关重要的。如果制定得正确，它们会消除摩擦。如果制定得不好，或者被忽视，我们就会看到真正的摩擦和争论，这可以驱走大多数友善的人，留下要对着火的房子负责的爱争论的核心人物。我一直试图对 ØMQ 和以前的社区做的一件事是创建可重用的规则，这或许意味着我们不太需要律师。
- 第三，社区需要某种形式的财政支持。这是打破大多数船只的锯齿状岩石。如果你对一个社区投入过少，它变得更富有创造性，但核心贡献者会心力交瘁。如果你投太多的钱进去，你吸引到的是那些从来不说“不”的人才，而社区失去了它的多样性和创造性。如果你创建一个基金让人们分享，他们会（恨恨地）为它争斗。对于 ØMQ，我们（iMatix）花时间和金钱来进行营销和包装（比如这本书），并提供诸如 bug 修复、发布和网站等基本的照管。
- 335 最后，销售和商业调解是重要的。专家撰稿人和客户之间有天然的市场，但两者都在互相交流上有点不称职。客户认为因为该软件是免费的，所以支持也应该是免费或非常便宜的。而贡献者都羞于为他们的工作要求一个公平的回报率。这使得市场营销是困难的。我的工作（和我的公司的利润）的一个增长的部分是简单地把希望得到帮助的 ØMQ 用户与能够提供这种帮助的社区专家撮合起来，并确保双方都得到满意的结果。

我见过具有高尚目标的杰出人物的社区，因为创始人做错了这四样东西中的一些或全部而倒闭。核心问题是，你不能期望任何一个公司、个人或群体提供一贯出色的领导。在今天起作用的东西往往明天不会起作用，而随着时间的推移，结构变得更加牢固，而不是更加灵活。

我能找到的最佳答案是把如下两件事情组合起来。首先是 GPL 及其保障的可结合性。无论授权者有多么糟糕，无论它有多么试图私有化并攫取社区的工作成果，如果某个软件是 GPL 许可的，那么这项工作就可以离开并寻找更好的授权者。在你说，“这都是开源提供的”之前认真地考虑它。我可以通过聘用核心贡献者和不发布任何新的补丁来扼杀一个 BSD 许可的项目。但即使有十亿美元花费，我也不能干掉一个 GPL 许可的项目。二是授权者的哲学上的无政府主义模型，是我们选择授权者，而它不拥有我们。

ØMQ 过程：C4

当我们说 ØMQ 时，有时表示的是 `libzmq` 核心库。在 2012 年初，我们将 `libzmq` 过程合成到一个正式的合作协议中，我们称之为集体代码建设合同（*http://rfc.zeromq.org/spec:16*），或 C4。你可以认为这是 GPL 之上的一层。事实上，`libzmq` 并不完全坚持 C4，因为历史的原因，我们用 Jira 来代替 GitHub 的议题跟踪器。除此之外，这些都是我们的规则，我将解释每个规则背后的理由。

C4 是 GitHub 派生 + 提取模型（*https://help.github.com/send-pull-requests/*）的演变。你可能会感觉我是 Git 和 GitHub 的狂热爱好者。确实如此：在过去几年，这两个工具已经对我们的工作产生如此积极的影响，特别是当涉及建设社区时。

语言

本文档中的关键字“必须”、“不得”、“要求”、“应当”、“不应”、“应该”、“不应该”、“建议”、“可以”、“可选的”是按照在 RFC 2119 中的描述解释的。

（译者注：RFC 2119 的地址为 *http://www. ietf.org/rfc/rfc2119.txt*。规定表示要求的动词：

- MUST，必须的。通过它描述的对象，是强制要求的。它与 REQUIRED 和 SHALL 含义相同。
- MUST NOT，不允许的。通过它描述的对象也是强制的。与 SHALL NOT 同义。
- SHOULD，在通常情况下，意为应当这样。但是，特殊情况下除外。与 RECOMMENDED 同义。
- SHOULD NOT，在通常情况下，意为不是这样。但是，特殊情况下除外。与 NOT RECOMMENDED 同义。
- MAY 可选的描述对象。与 OPTIONAL 同义。

在本书译文中，为了区分原来的英文单词，将“MUST”，“MUST NOT”，“REQUIRED”，“SHALL”，“SHALL NOT”，“SHOULD”，“SHOULD NOT”，“RECOMMENDED”，“MAY”，“OPTIONAL”分别译为上述名称。）

336 从 RFC 2119 语言开始，C4 的文本让人很清楚，它打算作为一个协议，而不是一个随机的书面建议集。协议是各方之间的合约，它定义了各方的权利和义务。这里的各方可以是网络中的对等节点，也可以是在同一个项目上工作的陌生人。

我觉得 C4 是第一次有人试图把社区的规则手册作为一个正式的和可重复使用的协议规范来编纂的。此前，我们的规则分布在多个 wiki 页面中，在许多方面，它们是相当专门地用在 libzmq 上的。但经验告诉我们，规则越是正式、准确、可重复使用，它对前期合作的陌生人就越容易。而少一些摩擦则意味着一个社区更具扩展性。在 C4 的时候，对于我们在 libzmq 项目中正在用的是什么过程，也有一些分歧。不是每个人都感到受同样的规则束缚。远的不说，有些人觉得他们有一个特殊的地位，这会导致与社区的其他人产生摩擦。编纂规则使事情变得清楚了。

C4 非常易于使用：只需要在 GitHub 上托管你的项目，让另外一个人加入，并开放提取请求。在你的自述文件中，放置一个指向 C4 的链接就行了。我们已经在不少项目中这样做了，它确实能正常工作。将这些规则应用到我自己的工作，比如 CZMQ 中，我已经惊喜了好几次。我们中无人对此感到惊奇，以至于我们可以离开其他人独立开展工作。

目标

> C4 的目的在于为开放源码软件项目提供可重复使用的最佳合作模式。

编写 C4 的短期原因是为了结束在 libzmq 贡献过程中的争论，持异议者去了别处。ØMQ 社区发展是顺利和轻松的，正如我已经预言的。大多数人感到惊讶，但都很欣慰。C4 没有受到过真正的批评，除了其分支的政策，我以后会来讨论它，因为它确实值得专门讨论。

还有一个历史原因，我在这里回顾它：作为社区的创始人，你求人来投资于你的财产、商标和品牌。作为回报，而这就是我们对 ØMQ 所做的，你可以使用该品牌来设置一个质量的标杆。当你下载了有“ØMQ”标记的产品时，你知道，它是按照一定的标准生产的。下面是质量的一个基本规则：写下你的过程，否则你就不能改进它。我们的流程不是完美的，也永远不能够达到完美。但它们中的任何缺陷都是可以修复，并进行测试的。

因此，使得 C4 可重复使用是真正重要的。要对尽可能好的过程了解更多，我们就需要从最广泛的项目取得成果。

它具有以下具体目标：

> 337 通过减少对新贡献者的摩擦和用较强的正反馈创建扩展的参与模式，来最大限度地扩展围绕一个项目的社区的规模。

头号目标是使社区规模最大化并尽可能健康——不是技术质量，不是利润，不是性能，也不是市场份额。我们的目标只是增加对项目做出贡献的人的数量。这里的科学很简单：社区越大，结果就越准确。

> 要通过分离不同的技能来减轻对关键人物的依赖，以便在任何需要的领域有更大的能力池。

也许我们在 `libzmq` 中面临的最糟糕的问题是对可以同时理解代码，管理 GitHub 的分支，并制作干净版本的人的依赖性。这就像在寻找可以跑马拉松并且可以短跑、游泳，还能举重的运动员。我们人类在专业化方面真的擅长。要求我们真的擅长两个矛盾的东西就会急剧减少候选者的人数，而这对任何项目都是一件坏事。2009 年，在 `libzmq` 中有过这样严重的问题，我们通过将维护者的角色拆分为两个：一个人做补丁程序而另一个人做发布版本，从而解决了这个问题。

> 通过增加决策过程的多样性，使项目能够开发得更快、更准确。

这是理论上的，证据并不充分，但并非是捏造的。社区的多样性越大，并且可以加入讨论而不用担心被批评或辞退的人的数量越多，软件就会开发得越快，越准确。在这里，速度是相当主观的因素。在错误的方向走得很快不仅是无用的，它还是一种积极的破坏（在我们切换到 C4 之前，在 `libzmq` 中吃了不少苦头）。

> 通过允许安全的实验，快速出故障，以及隔离稳定的代码，支持项目从实验版本到稳定的版本的自然生命周期。

说实话，这个目标似乎已衰落到无足轻重的程度。它是这个过程相当有趣的效果：Git 的主版本几乎都是完全稳定的。这使得必须处理变化的规模及其延迟时间，即从某人编写代码到有人真正充分使用它相隔的时间长度。然而，人们仍然期望“稳定”的版本，因此我们将这个目标继续保留一段时间。

> 降低项目库的内部复杂性，从而使贡献者更容易参与，并减少错误的范围。

好奇的观察：成长于复杂情况的人喜欢建立复杂性，因为这使他们保持高身价。这是眼镜蛇效应（请在谷歌上查阅它）（译者注：该术语用于形容政治和经济政策下错误的刺激机制）。Git 使得分支简单，并给我们留下了司空见惯的说法，“一旦你了解了一个 Git 分支仅仅是一个折叠的五维轻子空间，它有一个分离的历史记录而没有中间缓存。Git 就是容易的。”开发者不应该在使用他们的工具时觉得自己很蠢。我已经看到太多的顶 338
级开发者在 Git 的分支上接受传统智慧时被存储库结构弄糊涂。亲爱的读者，我们很快就会再回过头来处理 Git 分支。

强制执行项目的集体所有制，从而增加对贡献者的经济诱因，并降低被敌对实体劫持的风险。

最终，我们是经济的动物，并且“我们拥有这个，我们的工作永远不能被用来对付我们”的意识使得人们更容易投资于像 ØMQ 这样的开源项目。并且它不能只是一种感觉，它还必须是真实的。要使集体所有制能正常工作，有许多方面的工作要做，当我们经历 C4 时，我们将逐项地看到这些工作。

热身

项目应当使用 Git 分布式版本控制系统。

Git 有它的缺点。其命令行 API 非常不一致，并且它有复杂且混乱的内部模型，这些都以最轻微的挑衅击打到你的脸上。但是，除了尽最大的努力让它的用户感到他们自己很笨外，Git 把它的工作做得非常非常好。更务实的，我发现，如果你远离某些领域（分支！），人们就能迅速地学习 Git，并且不会犯很多错误。这符合我的要求。

项目应当被托管在 github.com 或等价的网站上，这就是所谓的“平台”。

我敢肯定，有一天，会有大公司购买 GitHub 并破坏它，而另一个平台将出来接替它。GitHub 提供了近乎完美的一套最小的、快速而简单的工具。我已经向它抛出了数百人，而他们都像苍蝇叮在一盘蜂蜜上一样地坚守在上面工作。

项目应当使用平台的议题跟踪器。

我们在 `libzmq` 中犯了切换到 Jira 这个错误，因为我们还没有学会如何正确使用 GitHub 的议题跟踪器。Jira 是一个很好的例子，它说明了如何把有用的某个东西变成一个复杂得一塌糊涂的东西，因为业务依赖于销售更多的“功能”。但是，即使没有对 Jira 的批评，在同一平台上保留议题跟踪器也意味着少学习一个用户界面，少一次登录，以及在问题和补丁程序之间的平滑整合。

项目应该具有清楚地记录代码风格的指南。

这是一个协议插件：在这里插入代码风格指南。如果你不记录你使用的代码风格，那么除了偏见，就没有依据来拒绝补丁程序。

339 “贡献者”是希望提供补丁程序的人，此处的补丁程序是指解决一些明确问题的一套提交方案。

> “维护者”是将补丁程序合并到项目中的人。维护者不是开发者，他们的工作是执行过程。

现在我们进入到各方的定义，并对角色进行分解，以使我们摆脱因为对罕见的个别人才的结构性依赖而产生的罪过。这在 libzmq 中的工作效果很好，但正如你会看到的，它依赖于过程的其余部分。C4 不是随心所欲的自助餐，你将需要整个过程（或非常像它的某种东西），否则它就将无法维系下去。

> 贡献者不应有对存储库的提交权限，除非他们同时也是维护者。

> 维护者应当有对存储库的提交权限。

我们希望避免人们直接将他们的更改推给主版本。这是在 libzmq 历史上麻烦最大的来源：花了数月或数年才能完全稳定的大量原始代码。我们最终还是遵循类似 PyZMQ 的其他 ØMQ 项目使用的提取请求的方式。我们甚至走得更远，并规定所有变更都必须遵循相同的路径，“特别的人”也不例外。

> 每个人，没有高低贵贱歧视，都应当享有平等的权利成为本合同条款下的贡献者。

我们必须明确说明这一点。libzmq 维护者曾经仅仅因为他们不喜欢一些补丁程序而拒绝它们。现在，虽然这对库的作者听起来是合理的（虽然 libzmq 不是由任何一个个人编写的），但是请记住我们的目标是创建由尽可能多的人拥有的工作成果。如果有人说：“我不喜欢你的补丁程序，所以我要拒绝它”，这等于说：“我声明我对这个软件拥有所有权，我觉得我比你强，我不相信你。”对于想成为你的共同投资者的人，这些都是有毒的讯息。

我认为个人的专业知识和集体智慧之间的这场斗争也在其他领域上演着。它确立了维基百科，并且仍然在发挥作用，十年以后群体的工作超越了一小队专家建造的任何事情。对于我来说，我们通过合成知识来做软件，就像我们编写维基百科文章一样。

许可和所有权

> 项目应当使用 GPLv3 或其变体（LGPL、AGPL）。

我已经解释了充分混合能力如何创造了更好的规模，以及为什么 GPL 协议及其变种似乎对可重新混合的软件是最优的合同。如果你是一家大型企业，旨在在市场上倾倒代码，你不会想要 C4，但随后你也就不会真正关心社区了。

> 对项目源代码的全部贡献（“补丁程序”），都应当使用与项目相同的许可证。 340

这消除了对补丁程序任何特定的许可或贡献协议的需要。你对 GPL 代码执行派生，你在 GitHub 上发布你的重新组合版本，而你或其他人可以再将它作为源代码的一个补丁程序提交。BSD 不允许这样。任何包含 BSD 代码的工作也可能包含未授权的专有代码，所以你需要先有代码作者的明确指示，然后才能对它重新组合。

> 所有的补丁程序都由他们的作者拥有。不应有任何版权转让的过程。

下面我们就来说明人们相信他们在 ØMQ 的投资的关键原因：通过购买版权来创建一个封闭源代码的 ØMQ 竞争对手，这在逻辑上是不可能的。iMatix 也做不到这一点。而发送补丁程序的人越多，这就变得越难。ØMQ 不仅今天是自由和开放的，这种特定的规则意味着它会永远如此。请注意，并不是所有的 GPL 项目都是这样的，其中许多仍然要求将版权转让回维护者。

> 项目应当由它所有的贡献者集体拥有。

这也许是多余的，但值得一说：如果每个人都拥有自己的补丁程序，那么全部结果也由每一个贡献者拥有。拥有代码行并没有法律上的概念：这里的“工作成果”是至少一个源文件。

> 每个贡献者都应当负责在项目贡献者列表中确认自己。

换句话说，维护者不是因果报应会计师。任何想要荣誉的人必须自己亲自申请它。

对补丁程序的要求

在本节中，我们定义了贡献者的义务：具体而言，什么是“有效的”补丁程序，使维护者有章可循，他们可以根据它来接受或拒绝补丁程序。

> 维护者和贡献者必须拥有一个平台账户，并应当使用他们的真实姓名，或一个众所周知的别名。

在最坏的情况下，如果有人提交了有毒的代码（专利的，或由别人所拥有），我们需要能够跟踪谁在什么时候提交的，以便我们可以删除该代码。要求实名或一个众所周知的别名是降低假补丁程序的风险的理论策略。我们不知道这实际上是否可行，因为我们还没有遇到这个问题。

341

> 补丁程序应当正好是对一个确定和认可的问题的最小和准确的解决方案。

这就实现了我将在本章后面讲述的简单的面向设计的过程。一个清晰的问题，一个最小的解决方案：应用、测试并重复。

> 补丁程序必须坚持项目的代码风格指南，如果这些已被定义。

这仅仅是理智。出于本能的要求，我已经花时间清理了其他人的补丁程序，因为他们坚持把“else”写在“if”的旁边，而不是正下方。一致的代码是更为健康的。

> 补丁程序必须坚持下面定义的“公共合同进化”指导原则。

啊，痛，痛。我说的不是我八岁时，踩在凸起在一块木板上的 4 英寸长的钉子上的情况。那时的疼痛我相对可以忍受。我讲的是 2010-2011 年，我们拥有多个并行版本的 ØMQ，每个版本使用不同的不兼容 API 或线路协议的情况。这是一个执行空洞的坏规则的练习。该规则是，“如果你改变了 API 或协议，你应当创建一个新的主要版本”。与此相比，我宁愿给我脚上钉钉子，因为它的伤害更小。

我们在 C4 中所做的一项巨大变化是彻底禁止对这种破坏的容忍。令人惊讶的是，它居然不是很难。我们只是不容许破坏现有的公共合同，就是这样，除非每个人都同意，在什么情况下不这样。正如 Linus Torvalds 于 2012 年 12 月 23 日所说的名言那样：“我们不破坏用户空间！”

> 补丁程序不应包括来自其他项目的重要代码，除非贡献者是该代码的原作者。

此规则有两个作用。首先，它迫使人们做出最小的解决方案，因为他们不能简单地大片导入现有代码。在我见过的在项目中发生这种事的情况中，除非导入的代码是很清楚地分离的，否则总是带来坏的结果。第二个效果是，它避免了许可证争论。如果是你编写了补丁程序，你就被允许将其发布为 LGPL，而我们可以将它合并回去。但是假如你在网上找到一个 200 行的代码片段，并尝试粘贴它，我们将会拒绝。

> 补丁程序必须能够至少在最重要的目标平台上彻底地编译。

这可能是很高的要求，因为大多数贡献者只在一个平台上工作。

> “正确的补丁程序”是一个满足所有上述要求的补丁程序。

万一它是不明确的，我们又回到了法律术语和定义上来。

开发过程 342

在本节中，我们的目标是循序渐进地描述实际的开发过程。

> 对项目的更改应当以找准问题和对这些问题应用最小和准确的解决方案的模式来办理。

这是经过 30 年的软件设计经验获得的无歉意的强制措施。这是一个深刻而简单的设计方法：为实际问题制作最小和准确的解决方案。仅此而已。注意：在“准确”上的压力，这是一种罕见但基本的成分。在 ØMQ 中，我们没有功能要求。将新功能视为与错误一样的东西使新手感到困惑。但这个过程效果很好，并且不仅仅在开源领域。使用每一个单独的更改来阐述我们正在试图解决的问题，是确定变更是否值得做的关键。

> 开始更改前，用户应当在项目平台议题跟踪器上登记问题。

这是为了防止我们进入离线状态并防止无论是我们自己还是他人在壁垒中工作。虽然我们倾向于接受有明确的论证的提取请求，但这个规则让我们对令人混淆或过大的补丁程序喊“停”。

> 用户应当通过描述他们面对或观察的问题来编写议题。

“问题：我们需要的功能 X，解决方案：制作它”不是一个好议题。“问题：除非通过使用一个复杂的解决方法，否则用户不能执行常规任务 A 或 B。解决方案：制作功能 X”是一个体面的解释。因为我曾经合作过的每个人都有学习这个的需要，所以似乎值得重申：首先记录实际问题，其次是解决方案。

> 用户应当在他们的观测精度和解决此问题的价值上寻求共识。

因为很多明显的问题是虚幻的，通过明确地说明问题，我们给了别人一个机会来纠正我们的逻辑。“你只能大量地使用 A 和 B，这是因为函数 C 不可靠。解决方案：让函数 C 正常工作”。

> 用户不应登记功能要求、创意、建议，或者没有明确记载和证明的问题的任何解决方案。

不能登记创意、建议，或功能要求有几个原因。根据我们的经验，这些只是堆积在议题跟踪器中，直到有人将其删除为止。但更深刻的是，当把所有更改都视为问题的解决方案时，我们可以显而易见地优先考虑。无论是现实的，有人想现在就来解决的问题，还是不在台面上的问题。因此，愿望清单是不予考虑的。

> 343 因此，项目的发布历史应当是登记并解决了的有意义的议题列表。

我喜欢 GitHub 的议题跟踪器简单地列出所有我们在每个版本中解决了的议题。现在，我们仍然需要手工编写它。如果一个人在每个提交中都标上议题编号，并且如果他使用 GitHub 的议题跟踪器，（遗憾的是，我们还没有对 ØMQ 这么做），那么版本的历史就更易于机械地生成。

> 为了针对一个议题开展工作，一个贡献者应当对项目库进行派生，然后在他们的派生存储库上开展工作。

下面我们讲解 GitHub 分支 + 提取的请求模式，以便想要做贡献的新手只需学习一个过程（C4）。

> 要提交一个补丁程序，贡献者应当创建一个指回该项目的平台提取请求。

GitHub 已经将这个做得非常简单，我们不需要学习 Git 命令来做到这一点，对此我深表感激。有时候，我会对我并不特别喜欢的人们说，命令行 Git 是真棒，他们需要做的只是在试图将它用于实际工作之前，详细了解 Git 的内部模型。当数个月后我再次看到他们时，他们看起来 ...... 变样了。

> 贡献者不应将更改直接提交到项目中。

提交补丁程序的任何人都是贡献者，所有贡献者遵循相同的规则。没有给原作者的特殊权限，因为否则我们就不是在建设社区，而只是在增加我们的自负。

> 要讨论一个补丁程序，人们可以在平台提取要求、提交，或其他地方发表意见。

如果你是第一次走近，随机分布的讨论可能会造成混淆，但 GitHub 为目前所有的参与者解决了这个问题，它采取的办法是发送电子邮件给那些需要跟踪事情进展的人们。我们在 Wikidot 上有相同的经历，并采取相同的解决方案，并且它的效果很好。没有证据表明，在不同的地方讨论有任何负面影响。

> 维护者应当采用平台界面来接受或拒绝一个补丁程序。

通过 GitHub 的 Web 用户界面工作表示提取请求会被登记为议题，带有工作流程和讨论。我敢肯定还有更复杂的工作方式。复杂性是容易的，而简单性是极其困难的。

> 维护者不应接受自己的补丁程序。

多年前我们在 FFII 定义了一个规则来阻止人们的倦怠：任何项目不能少于两个人。单人 344
项目往往在眼泪中收场，或者至少是苦寂。我们对于倦怠，它为什么会发生，以及如何防止它（甚至治愈它）有相当多的数据。我将在本章后面探究这一点，因为如果你使用开源项目或在开源项目中工作，你就需要注意风险。“不合并你自己的补丁程序”的规则有两个目标。首先，如果你希望你的项目符合 C4 的认证，你就必须得到至少一个其他人提供的帮助。如果没有人愿意帮助你，也许你需要重新考虑你的计划。其次，对每个补丁程序都进行控制使得它更令人满意，让我们更加专注，并防止我们因为在赶时间，或者只是感觉慵懒而破坏规则。

维护者不应对正确的补丁程序进行价值判断。

我们已经说过这一点，但它值得重复：维护者的作用不是判断一个补丁程序的实质内容，而只是它的技术质量。补丁程序的实质性价值只随着时间的推移而出现：人们使用它并喜欢它，或者不使用它。如果某个补丁程序没有人使用，它最终会惹恼别人，并被人删除，也不会有人会为此而抱怨。

维护者应当迅速地合并正确的补丁程序。

有一个标准，我称之为*更改延迟*，这是指从确定一个问题到测试它的解决方案的往返时间。解决问题的速度越快越好。如果维护者无法像人们期望的那样迅速地响应提取请求，他们就不够称职（或者他们需要更多的帮手）。

为某个议题制作了一个提取请求后，贡献者可以将此问题标记为“就绪”。

默认情况下，GitHub 提供了通常的各种议题，但 C4 不使用它们。相反，我们只需要两个标签，“紧急”和“就绪”。如果一个贡献者愿意另一个用户来测试某个议题，就可以将它标记为“就绪”。

建立议题的用户应该在检查补丁程序是成功的之后关闭此议题。

当某人开启了一个议题而其他人对它开展工作时，最好是让原来的人来关闭议题。这可以充当为这个问题得到妥善解决的验证检查。

维护者应当要求改善不正确的补丁程序，并且如果贡献者不积极回应，那么应当拒绝不正确的补丁程序。

起初，我觉得所有补丁程序都值得合并，无论它有多么差。这里包含诱使的因素：我觉得，接受甚至明显虚假的补丁程序可以拉拢更多的贡献者。但是人们会对此感到不舒服，所以我们定义了“正确的补丁程序”的规则，而维护者的职责是检查质量。在消极方面，
345 我觉得我们并没有采取一些本来可以有的让更多的参与者有回报的有趣风险。在积极方面，这导致了 ØMQ 主版本（以及所有使用 C4 的项目）在几乎所有的时间都是一个具备生产质量的实用产品。

对一个正确的修补程序拥有价值判断的任何贡献者都应当通过自己的补丁程序表达这一点。

从本质上说，这里的目标是让用户试用补丁程序，而不是花时间争论其利弊。与制作一个补丁程序一样容易，使用其他补丁程序来还原它是容易的。你可能会认为这将导致“补丁程序战争”，但是这并没有发生。在 `libzmq` 中，我们只在极少数案例中遇到其中一个

贡献者制作的补丁程序被另一个觉得这个实验不在正确的方向的人清除的情况。这种方法比寻求前期的共识更容易。

> 维护者可以将对非源文件的更改直接提交到项目中。

此出口允许正在制作发行说明的维护者推送那些文档而无须创建一个议题，这将进而影响发行说明，导致对时空结构的压力，并可能不由自主地把时间倒流回到发明冰镇啤酒之前。这令人不寒而栗。同意发行说明不改变软件本身，这是更简单的。

建立稳定的版本

对于生产系统，我们希望对其稳定性有一些保障。在过去，这意味着取得不稳定的代码，然后对它进行几个月的精心调试，解决找到的错误和故障，直到它可以被安全地信任。多年来，iMatix 的工作，一直是对 `libzmq` 采取这种做法，即通过只允许 bug 修复，而没有新的代码加入到“稳定分支”的办法将原始代码转到包中。这是令人惊讶的，它并不像听起来那么吃力不讨好。

虽然我们使用 C4 过程全速运行，但我们发现，大部分的时间，`libzmq` 的 Git 的主版本大多是完美的。这将我们的时间释放出来去做更多有趣的事情，比如在 `libzmq` 之上构建新的开源层。然而，人们仍然希望得到这样的保证：许多用户根本不会去安装除了“官方”版本以外的版本。因此，稳定版本今天涉及两件事情：第一，在有一段时间没有出现新的变化，也没有任何戏剧性的开放 bug 的时候取得的主版本的快照，第二，一种对该快照进行微调来解决剩余的严重问题的办法。

这是我们在本节讲解的过程。

> 项目应当有一个始终保持最新的在建版本，并且应该始终在构建的分支（“主分支”）。

这种提醒是多余的，因为每一个补丁程序始终在构建，但它值得重申。如果主版本没有在构建（并通过其测试），那么有人需要察觉这一点。

> 该项目不应以任何理由使用主题分支。个人派生可以使用主题分支。 346

我很快就会来讨论分支。简言之（或“tl;dr”，译者注：太长了，读不下去（Too Long; Didn't Read），正如他们在网上所说的），分支使存储库过于复杂和脆弱，并且它们需要预先的协议，所有这些都是昂贵和可避免的。

> 为了制作一个稳定的版本，某人应当通过复制存储库对其进行派生，并从而成为这个存储库的维护者。

> 为稳定化而对项目派生可以单方面地且不经项目维护者的同意就进行。

它是免费软件。没有人垄断它。如果你认为维护者不能正确地产生稳定的版本，那就对存储库执行派生并自己来做这件事。派生不是一种失败，这是竞争的必备工具。你不能用分支做到这一点，这意味着一个基于分支的版本政策给项目维护者一种垄断。这是不好的，因为这比起如果有真正的竞争正在紧追着他们，会使他们变得更加懒惰和傲慢。

> 稳定项目的维护者应当通过提取请求来维护它，这些请求可以从派生的项目精选补丁程序。

或许 C4 的过程应该只是说，稳定化的项目像任何项目一样，都有维护者和贡献者。这就是这个规则的含义。

> 对于一个宣称“稳定”的存储库的补丁程序应当附有一个可重复的测试案例。

当心一个放之四海皆准的过程。新代码不要求与人们相信可在生产中使用的代码有相同的偏执狂。在正常的开发过程中，我们没有提到测试用例。这有一个原因。虽然我爱可测试的补丁程序，但很多更改是不容易（或者根本不能）测试的。然而，要稳定你要修复严重错误的代码库，以及你希望百分之百地肯定每一个变化都是准确的。这意味着对每次更改进行更改之前和之后的测试。

> 一个稳定的存储库的进展应该通过下面这些阶段：“不稳定”、“候选”、“稳定”，然后“遗产”。也就是说，稳定的存储库的默认行为是死亡。

这可能是过于详细的。这里的关键点是，这些派生的稳定存储库在最后都死了，因为主版本在不断地发展，并为了生产版本的发布而不断地被派生。

347 公共合同的演变

我说的“公共合同”的含义是 API 和协议。直到 2011 年年底，`libzmq` 的自然愉快的状态被失信和损坏的合同破坏了。我们完全停止了对 `libzmq` 的承诺(又名“路线图”)，现在，我们主导更改的理论是：它随着时间的推移认真和准确地出现。在 2012 年的芝加哥用户聚会中，Garrett Smith 和 Chuck Remes 将此称为“醉酒蹒跚到伟大”，这是我现在想起它来的原因。

我们仅仅通过禁止这种做法停止破坏公共合同。在此之前，只要我们改变了主版本号，破坏 API 或协议曾经是“OK”的（如，我们这么做了，每个人都痛苦地抱怨，我们忽略它们）。这听起来不错，直到你面临了同时让 ØMQ 版本 2.0、3.0 和 4.0 都在开发中，而且互不交流的情况。

所有公共合同（API 或协议）都应该记录在案。

你会认为这是对专业的软件工程师理所当然的事情，但情况并非如此，它不是。所以，这是一条规则。如果你想为你的项目取得 C4 认证，就要确保你的公共合同被记录在案，没有“这已在代码中指定”的借口。代码不是一个合同。（是的，我打算在某个时候创建一个 C4 认证过程来充当一个开源项目的质量指标。）

所有公共合同都应当使用语义版本控制。

这条规则主要是应人们要求创建的。我对它并没有真正的好感，因为语义版本化导致了所谓的“为什么 ØMQ 不与自身交流！？”这种灾难。我从来没有见过这解决了什么实际问题。它是一些有关库版本的运行时验证，或者一些诸如此类的东西。

所有公共合同都应该具备可扩展性和实验的空间。

现在，实际的情况是，公共合同确实在改变。这里说的不是关于不改变它们，而是关于安全地改变它们的事情。这意味着教育设计师（尤其是协议的设计师）要预先创建出那个空间。

修改了公共合同的补丁程序不应该破坏现有的应用程序，除非预先对这样做的价值存在共识。

有时补丁程序修复的是没有人使用的坏 API。这是我们需要的一个自由，但它应该是基于共识而不是一个人的教条。在 ØMQ v3.x 中，我们也受益于将 `ZMQ_NOBLOCK` 重命名为 `ZMQ_DONTWAIT` 吗？当然，这距离 POSIX 套接字的 `recv()` 调用更接近了，但是这值得破坏成千上万的应用程序吗？从来没有人把它作为一个问题报告。错误地引用理查德 · 斯托曼的话：“你创造一个理想世界的自由在离我的应用程序一英寸的地方停止了”。 348

往一个公共合同中引入了新功能的补丁程序应该使用新名称做这件事。

我们在 ØMQ 中经历了一次或两次新功能使用旧名称的情况（或者更糟，使用仍然在其他地方使用的名称）。ØMQ v3.0 中有一个新推出的“ROUTER”套接字，这是与 ØMQ v2.x 现有的 ROUTER 套接字完全不同的。究其原因：很显然，即使聪明的人有时也需要规则，以阻止他们做愚蠢的事情。

旧名称应该以系统的方式被废弃，这通过把新的名称标记为“试验品”，直到它们是稳定的，然后把旧名标记为“弃用”来实现。

这个生命周期表示法大有裨益，它实际上已经用一个一致的方向告诉了用户到底是怎么回事。“实验”的意思是“我们引入这个功能，并打算如果它正常工作就将它做成稳定的”。

这并不意味着，“我们已经引入了这个功能，并会在我们感觉喜欢它的任何时候删除它”。人们假设生存不止一个补丁程序周期的代码是应该存在的。“弃用”的意思是“我们已经取代了这个，并打算将其删除”。

> 当已经过去了足够长的时间，旧的弃用名称应该被标为“遗产”，并最终将被删除。

从理论上讲，这给了应用程序时间来没有风险地移动到稳定的新合同。你可以先升级，确保一切正常工作，然后，随着时间的推移，修正事物，以除去对弃用和旧的 API 和协议的依赖。

> 旧名字不应被新功能重新使用。

啊，是的，当 ØMQ v3.x 将最常用的 API 函数（`zmq_send()` 和 `zmq_recv()`）更名，然后再回收旧名称并把它们用于根本不相容的新方法（而且我怀疑很少有人真正使用）是有趣的。你应该再次将自己拍糊涂，但这就是实际发生的事，而我和任何人一样有罪。毕竟，我们没有更改版本号！这种经验的唯一好处就是得到这条规则。

> 当旧的名字被删除时，如果应用程序使用了它们，其实现必须引发一个异常（断言）。

我还没有测试过这条规则以确定它一定是有意义的。也许，它的意思是“如果你不能引发一个编译错误，因为 API 是动态的，那么就引发一个断言。”

349 C4 是不完美的。没有东西是完美的。更改它的这个过程（Digistan 的 COSS）现在是有点过时了：它依赖于一个单人的编辑工作流以及派生而不是合并的能力。这似乎能够工作，但像 C4 这样的协议可能最好使用 C4。

一个实际例子

在“XPUB 订阅通知”的电子邮件主题中（*http://lists.zeromq.org/pipermail/zeromq-dev/2012-October/018838. htm*），Dan Goes 询问如何制作一个了解一个新的客户端何时订阅和发送前面匹配的消息的发布者。这就是被称为“最后一个值缓存”的标准发布-订阅技术。在类似 pgm 的单向运输协议中(其中订阅者根本没有把数据包发回给发布者)，这不能做到。但是，通过 TCP，如果我们使用一个 XPUB 套接字，如果该套接字没有巧妙地过滤掉重复的订阅，以减少上行流量，这是可以做到的。

虽然我不是 `libzmq` 的一名专家级贡献者，但也看得出来这似乎是一个有趣的要解决的问题。这能有多难？我先将 `libzmq` 存储库派生到我自己的 GitHub 账户，然后将其克隆到

我的笔记本电脑，我在那里构建它：

```
Git clone git@github.com:hintjens/libzmq.git
cd libzmq
./autogen.sh
./configure
make
```

因为 `libzmq` 代码整洁并且组织良好，所以找到需要修改的主文件（*xpub.cpp* 和 *xpub.hpp*）是很容易的。每个套接字类型都有其自己的源文件和类。它们继承自 *socket_base.cpp*，这个文件里面有特定于套接字选项的这个钩子：

```
//  首先，检查特定的套接字类型是否重载了这个选项。
int rc = xsetsockopt (option_, optval_, optvallen_);
if (rc == 0 || errno != EINVAL)
    return rc;

//  如果套接字类型不支持该选项，则将它传递给通用选项解析器
return options.setsockopt (option_, optval_, optvallen_);
```

然后我检查 XPUB 套接字是在哪里过滤掉重复的订阅的，检查结果是在其 `xread_activated()` 方法中：

```
bool unique;
if (*data == 0)
    unique = subscriptions.rm (data + 1, size - 1, pipe_);
else
    unique = subscriptions.add (data + 1, size - 1, pipe_);

//  如果订阅不是一个重复项，则保存它，以便它可以被
//  传递给下一次的 recv 调用使用
if (unique && options.type != ZMQ_PUB)
    pending.push_back (blob_t (data, size));
```

350

在这个阶段，我不太关心 `subscriptions.rm()` 和 `subscriptions.add()` 的工作细节。该代码似乎是显而易见的，除了"订阅"也包括退订，这困惑了我几秒钟。如果在 `rm` 和 `add` 方法中有别的奇怪的东西，这是一个留待以后解决的单独问题。现在是时候对这个更改制作一个议题了。我首先打开 *https://zeromq.jira.com* 网站，登录，并创建一个新的条目。

Jira 亲切地给我提供了"错误"和"新功能"之间的传统的选择，而我花了 30 秒来思考这种适得其反的历史区别是从哪里来的。据推测，"我们会免费修复 bug，但你要为新功能付费"的商业建议，这源于"你告诉我们你想要什么，我们会为 $X 做这个功能"的

软件开发模式，而这一般会引出“我们花了三倍于 $X 的钱，但我们得到了什么？！”之类怒斥的电子邮件。

把这样的想法放在一边，我创建了一个议题，#443（*https://zeromq.jira.com/browse/LIBZMQ-443*），并描述了这个问题和可能的解决办法：

> 问题：XPUB 套接字过滤掉重复的用户（故意设计的）。然而，这使得它不可能做到基于订阅的智能。见位于 *http://lists.zeromq.org/pipermail/zeromq-dev/2012-October/018838.html* 的用例。
>
> 解决方案：给这种行为配置一个套接字选项。

然后，到了对它命名的时间。该 API 位于 *include/zmq.h*，所以这是我添加选项名称的地方。当你发明在一个 API 中或任何地方的一个概念时，请花一点时间来选择一个明确、简短和明显的名称。不要投靠需要额外的上下文来理解的通用名称。你有一个机会来告诉读者你的概念是什么以及能做什么。类似 ZMQ_SUBSCRIPTION_FORWARDING_FLAG 的名称是可怕的。从技术上说，这是在正确的方向上的一种目标，但它悲惨地冗长且难懂。我选择 ZMQ_XPUB_VERBOSE：简短而明确，并明确了 on/off 开关，其中默认设置是“off”。

接下来，该是将私有属性添加到 *xpub.hpp* 的 xpub 类的定义中的时候了：

```
//  如果为 true，则将所有上游订阅消息都发送出去，而不仅仅是那些唯一的订阅消息
bool verbose;
```

然后从 *router.cpp* 选取一些代码来实现 xsetsockopt() 方法。最后，我修改了 xread_activated() 方法来使用这个新的选项，而当我在修改这个方法时，也令套接字类型上的这个测试更加明确了：

```
//  如果订阅不是一个重复项，则保存它，以便它可以被
//  传递给下一次的 recv 调用使用
if (options.type == ZMQ_XPUB && (unique || verbose))
    pending.push_back (blob_t (data, size));
```

351 它第一次构建得很好。这让我有点怀疑，但由于懒惰和时间问题，我没有立即做出测试用例来实际测试该更改。过程不要求那样做，即使通常我这样做只是为了捕获我们都不可避免犯下的 10%的错误。但是，我做了，将这个新选项记录在 *doc/zmq_setsockopt.txt* 手册页。在最坏的情况下，我不过添加了一个没什么用的补丁程序，但我肯定没有破坏任何东西。

我没有实现一个匹配 zmq_getsockopt() 的方法，因为“最小”是名副其实的。没有明显的用例来获取你大概只是在代码中设置的一个选项值。对称不是使补丁程序的大小加倍

的一个有效理由。我确实要记录下新的选项，因为该过程说："所有公共合同都应当记录在案。"

提交了该代码后，我再将此补丁程序推送到我派生出来的存储库（"origin"）：

```
Git commit -a -m "Fixed issue #443"
Git push origin master
```

切换到 GitHub 的 Web 界面，我进入我的 `libzmq` 派生存储库，按下那个在顶部的"提取请求"大按钮。GitHub 要求我提供一个标题，所以我输入"添加 ZMQ_XPUB_VERBOSE 选项。"我不知道为什么当我做了一个整洁的提交消息时，它会要求这个，但是，嘿，这里让我们顺其自然。

这制作了一个带有两个提交的可爱的小提取请求：其中一个提交我一个月前已经在发行说明中做了，以对 3.2.1 版本做准备（当你大部分时间都在坐飞机出差时，一个月过得很快），另一个提交在我对问题 #443 的修复（37 行新代码）中。GitHub 允许在你已经拉开提取请求的序幕后继续制作提交。它们会被一次性排队并合并。这使事情简化了，但维护者可能会因为某个软件包基于一个看起来无效的补丁程序而整体拒绝它。

因为 Dan 在等待此修复程序（至少在我非常乐观的想象中），然后我又回到了 *zeromq-dev* 邮件列表，告诉他我已经做了补丁程序，并附了一个到此提交的链接。我得到反馈越快越好。当我在韩国做这个补丁程序时是凌晨 1 点，所以这时在欧洲是傍晚，而在美国是早晨。当你与世界各地的人们一起工作时，你就学会了数时区。那时 Ian 是在一个会议上，Mikko 正在上飞机，而 Chuck 可能是在办公室，但 3 小时后，Ian 合并了提取请求。

在 Ian 合并提取请求后，我将我的派生与上游 `libzmq` 存储库再同步。首先，我添加了一个"remote"，告诉 Git 这个库位于哪里（只在我工作的目录添加一次）：

```
Git remote add upstream git://github.com/zeromq/libzmq.git
```

然后我从上游主分支将修改提取回来，并检查了 Git 的日志来验证：

```
Git pull --rebase upstream master
Git log
```

而为了给 `libzmq` 贡献补丁程序，人们需要学习并使用相当多的 Git 功能，6 个 Git 命令 352
和在网页上的一些点击。最重要的是，作为一名天生懒惰、愚蠢，且容易被弄糊涂的开发者，我不必学习 Git 的内部模型，而我从来不必去做涉及我们称之为"Git 分支"的结构复杂的地狱般的引擎的任何事情，接下来，那些 Git 分支企图暗杀我们。让我们生活在危险中！

Git 分支是有害的

Git 的最常用功能之一是它的分支。几乎所有使用 Git 的项目都使用了分支，而“最佳”分支策略的选择就像开源项目的一个成年仪式。Vincent Driessen 的 Git-flow（*http://nvie.com/posts/a-successful-git-branching-model/*）也许是最有名的。它有基础分支（主、开发）、主题分支、发布分支、热修补程序分支以及支持分支。很多团队都采用 Git-flow，甚至有 Git 的扩展来支持它。

下面是 C4 中的一部分，当你第一次读它的时候也许会感到震惊：

> 项目不应以任何理由使用主题分支。但个人派生可以使用主题分支。

需要明确的是，我说的是在共享存储库中的公共分支。对于私人的工作，如在不同的议题上的工作，使用分支的效果似乎足够好，但它比我个人喜欢的更复杂。再次引用 Stallman 的话：“你创造复杂性的自由，结束在距我们的共享工作空间一英寸的地方。”

如同 C4 的其余部分，针对分支的规则都不是偶然的。它们来自我们制作 ØMQ 的经验，这从 Martin Sustrik 和我重新思考如何制作稳定的版本开始。我们都喜爱并欣赏简单性(有些人似乎对复杂性有很大程度的容忍)。我们聊了一会儿 …… 我问他，“我要开始做一个稳定的版本，让我在你正在工作的 Git 上制作一个分支好吗？”Martin 不喜欢这个主意。“好吧，如果我派生这个存储库，那么我就可以从你的存储库将补丁程序移动到那一个存储库。”那样让我们俩都感觉好多了。

ØMQ 社区的许多人对此的反应是震惊和恐惧。人们觉得我们太懒惰了，并使贡献者需要更加努力地去寻找“正确”的存储库。不过，这似乎很简单，实际上它工作得很顺利。其中最好的部分是，我们每个人都按自己的意愿开展工作，然而在 ØMQ 库已经感到极端复杂（而且它甚至不是类似 Git-flow 的任何东西）前，这都是感觉简单的。并且它的效果很好。唯一不足的是，我们失去了一个统一的历史记录。现在，历史学家也许会觉得被抢劫了，但老实说，对于谁在什么时候在包括所有的分支和实验版本中修改了什么的历史细节，我看不出它们会造成任何显著的痛苦或摩擦。

353 人们已经习惯于在 ØMQ 中的“多库”的做法，我们已经非常成功地将它运用在其他项目中。我自己的看法是，历史会做出判断，Git 分支和类似 Git-flow 的模式是从 Subversion 和单片存储库的日子继承的臆想问题的一个复杂的解决方案。

更深刻的，并且也许这就是为什么大多数人似乎是“错误的”：我认为“分支与派生”的争论真的是一个更深层次的关于如何使软件最优地“设计与进化”的争论。我将在下一节解决那个更深层次的争论。而现在，我会通过看一些标准，并在每一个标准中对分支和派生加以比较，尽量做到科学看待我对于分支的非理性的仇恨。

简单性与复杂性的对比

越简单越好。

分支比派生更复杂没有内在原因。然而，Git-flow 使用五种类型的分支，而 C4 使用两种类型的派生（开发和稳定）和一个分支（主分支）。从而间接地证明，分支导致比派生更复杂。对于新用户来说，学习使用许多个除了主版本外，没有其他分支的库，绝对是更容易的，并且我们已在实践中对此进行了测量。

更改延迟

交付越小且越快速就越好。

开发分支似乎与庞大、慢速、有风险的交付强烈关联。“对不起，我必须在我们可以测试新版本之前合并这个分支”标志着过程的崩溃。这当然不是 C4 的工作方式，C4 的方式是紧密专注于个别问题及其最小的解决方案，允许在开发中分支会引起更改延迟。派生的结果却不同：它由派生者来确保他的更改被干净地合并，并让它们保持简单，因此它们不会被拒绝。

学习曲线

学习曲线越平滑越好。

绝对的证据表明，Git 分支的使用学习起来很复杂。对于一些人来说，这没什么。不同的人曾经好几次告诉我，我之所以不喜欢分支，是因为我“从来没有正确地学会 Git”，这是公平的，但这是对工具的批评，而不是对人类的批评。

出故障的成本

出故障的成本越低越好。

分支要求开发者更完美，因为错误可能会影响他人。这引起出故障的成本。派生使得故 354
障非常廉价，因为从根本上说，发生在一个派生中的东西不可能影响不使用该派生的其他人。

前期协调

对前期协调的需要越少越好。

你可以做一个敌对的派生，但你不能做一个敌对的分支。分支依赖于前期的协调，这是昂贵和脆弱的。一个人可以打消整个团队的欲望。例如，在 ØMQ 社区中，我们曾经就 Git 分支模型无法达成一致意见长达一年时间。我们通过使用派生取代分支解决了那个问题。该问题就消失了。

可扩展性

项目的可扩展性越高越好。

在所有分支策略中的强假设是，存储库是项目。但是，在一个存储库中，你可以限制一起工作的人数。正如我所解释的，前期协调成本可能成为致命的问题。一个更为现实的项目，它通过允许任何人开启自己的存储库，并确保这些存储库可以一起工作来扩展。一个类似 ØMQ 的项目有几十个存储库。派生看起来比分支更具可扩展性。

惊奇和期望

惊奇越少越好。

人们期待分支，并且发现派生是不寻常的，因此感到迷惑。这是分支取胜的一个方面。如果你使用分支，同一个补丁程序会有相同的提交散列标签，而在跨派生时，该补丁程序将有不同的散列标签。这使得在跨派生时难以跟踪补丁程序，这是实情。但严肃地说，必须跟踪十六进制散列标签不是一个功能，这是一个错误。有时候，更好的工作方法，只是在一开始令人惊讶。

参与的经济学

奖励越有形越好。

人们喜欢拥有自己的工作，并获得荣誉。使用派生比使用分支更容易做到这一点。派生以健康的方式创造更多竞争，而分支打压竞争对手，并迫使人们进行协作并共享荣誉。这可能听起来是积极的，但以我的经验，这会让人失去动力。一个分支不是可以“拥有”的产品，而一个派生却可以是一个产品。

355 ## 在冲突中的强壮性

越能在冲突中生存的模型越好。

不管你喜欢与否，人们为了自我、地位、信仰，和世界的理论而斗争。挑战是科学的必要组成部分。如果你的组织模式取决于协议，你不会在第一次真正的斗争中生存。分支不能在真正的争论和斗争中生存，而派生可以是互相抵触的，但仍然使各方受益。这的确是自由软件的工作原理。

隔离的保证

生产代码和实验代码之间的隔离越强越好。

人都会犯错误。我曾见过将实验代码误推送到主线产品的情况。我曾见过人们在压力下制作出差劲的令人恐慌的更改。如果你可以把代码推送到随机分支 -X，你也可以把它推

送到主分支。分支不能保证对关键的生产代码的隔离，但派生能保证这种隔离。

能见度

工作的能见度越高越好。

派生有观察器、议题、自述文件和wiki，而分支没有这些。人们尝试派生，建立它们，破坏它们，修补它们。分支待在那里，直到有人记得对它们开展工作。派生有下载和压缩包，而分支没有。当我们寻求自我组织时，问题越明显和越具有声明性，我们就可以更快、更准确地工作。

结论

在本节中，我列出了一系列的论点，其中大部分来自于团队成员。它似乎可以归纳如下：Git的老手坚持认为分支是轻车熟路的，而新人当被要求浏览Git分支时往往会感到害怕。Git不是一个容易掌握的简单工具。偶然间，我们发现，当你完全停止使用分支时，Git的使用就会变得轻而易举。它最终归结为6个命令（克隆、远程、提交、记录日志、推送、提取）。此外，无分支的过程实际上效果很好，我们已经用它好几年了，除了使老手感到惊喜和跨多个存储库的“单一”项目在增长外，它没有明显的缺点。

如果你无法使用派生，也许是因为你的公司不信任GitHub的私有库，那么你或许可以使用主题分支，每个议题一个分支。然而，这么做你仍将遭受获取前期共识的成本、低竞争力和人为错误的风险。

为创新而设计

让我们来看看创新，维基百科将创新定义为“通过用增值新途径满足各种新的需求、难以言喻的需求，或者老客户和市场需求的解决方案来开发新价值。”这真的只是意味着更廉价地解决问题。这听起来非常简单，但倒塌的科技巨头的历史证明，事实并非如此。我将尝试解释团队如何经常把它弄错，并提出一个正确地创新的方法。 356

双桥传说

两位老工程师在谈论他们的生活和吹嘘他们最伟大的项目。其中一位工程师解释他是如何设计有史以来最伟大的桥梁之一的。

“我们横跨一个河谷建造了它”，他告诉他的朋友。“河谷既宽又深。我们花了两年时间研究选址，并选择设计师和材料。我们聘请最好的工程师，又花了五年来设计这座桥。我们找了最大的工程公司来承建结构、塔、收费站，以及将桥连接到主要高速公路的道路。

施工期间有数十人死亡。在道路层下面，我们有火车道并为自行车爱好者开辟了一条特殊的通道。这座桥代表了我数年的生活。”

第二位工程师想了一会儿，接着说。“有一天晚上，我和一个朋友去喝伏特加酒，我们都喝醉了，然后我们把一根绳子的一头抛过一个峡谷，”他说。“只要一根绳子，两头绑在峡谷两岸的两棵树上。有两个村庄，各在峡谷的一侧。起初，人们使用滑轮跨越绳子拉包裹。这时，有人扔了第二根绳索，并建造了一个步行道。虽然这是危险的，但孩子们很喜欢它。然后，一群男人重建了它，使它牢固，妇女开始每天带着她们的农产品跨越峡谷。一个市场就在桥的一侧发展起来，它慢慢变成了一个大镇，因为那里有一片很大的空地可以盖房子。索桥被替换为木桥，以便让马和马车穿越。然后，全镇建成一座带有金属横梁的真正的石桥。后来，他们将桥的石头部分替换为钢材，而今天在同一个地方有一座吊桥架在了那里。”

第一位工程师沉默了。“有趣的事情”，他说，“约 10 年后，我们建成的桥被拆除了。原来它建在了错误的地方，没有人想用它。有人在继续下行几英里的地方扔了一根绳子穿过峡谷，而这里也正是每个人都去的地方。”

ØMQ 的路线图是如何失去的

在 2012 年年初，在里昂的 Mix-IT 会议讲演 ØMQ 时，我被问了好几次“路线图”，我的回答是：不再会有任何的路线图。我们有过路线图，但我们删除了它们。我们不是试图
357 让少数专家为它制订出下一个步骤，而是允许这样的事情有机地发生。听众真的不喜欢我的答案。因为它如此的不法国、不传统、不地道、不浪漫。

然而，ØMQ 的历史很清楚地表明，为什么路线图是有问题的。在开始的时候，我们有一个小团队做这个库，有几个贡献者，并没有记录在案的路线图。随着 ØMQ 变得更加流行，我们切换到更多的贡献者，用户们要求提供路线图。所以我们将我们的计划收集在一起，并试图将它们组织为发行版本。在这里，我们写的，是在下一版本中会有的内容。

在我们推出了发布版本时，我们遇到了承诺某样东西很容易，而使其按计划进行相当困难的问题。一方面，许多工作是志愿的，而且到现在还不清楚你如何去强制志愿者去承诺一个路线图。而且，优先级可能随着时间变化而大幅度转移。所以我们做出了我们不能遵守的承诺，而真正的发布与路线图不匹配。

第二个问题是，通过定义路线图，我们实际上是在提出要求，使其他人更难以参加。人们确实喜欢贡献他们认为是自己的创意的更改。写下要做的事情的清单将贡献变成了一件苦差事，而不是一个机会。

最终，我们看到了 ØMQ 中的有些更改是非常令人痛苦的（例如，API 和协议中不兼容的更改），而路线图无助于解决这一点，尽管对“正确地做某事”有大量的讨论和努力。很显然，我们需要一种不同的方法来定义更改过程。

我想进一步探讨一个奇怪的讽刺，因为它支撑着 ØMQ 社区自 2012 年开始已采取的方向。

在创新的主导理论中，才华横溢的个人对庞大的问题集进行反思，然后小心和准确地创建一个解决方案。有时，他们有“得到”整个大问题集的精辟简单的答案的“尤里卡”时刻（译者注：希腊语“我知道了”，阿基米德发现王冠所含纯金量时的欢呼）。发明者和发明的过程是罕见的、珍贵的，并可以占垄断地位。历史充满了这样的英雄人物。

但是，随着研究更加深入，你会发现事实与上述理论并不相符。历史不会呈现单个的发明者：它呈现了幸运的人，他们窃取或声称拥有正在被许多人用在工作上的思路的所有权。它显示了杰出的人惊人地幸运了一次，然后就在毫无结果和毫无意义的任务上花费几十年。最知名的大发明家，像托马斯 · 爱迪生，实际上只是很擅长管理大团队完成的系统化的广泛研究。这就像声称史蒂夫 · 乔布斯发明了苹果公司生产的每种设备。这是一个很好的神话，良好的营销手段，但在实用科学中是完全无用的。

最近的历史，记录得更好并且更不容易被操纵，它很好地表明了这种情况。互联网肯定 358
是一个技术的最具创新性和飞速发展的领域，并且是记录得最好的领域之一。它没有发明者。相反，它具有许多经济型的人，他们仔细并逐步解决了一长串迫切的问题，记录他们的解决方案，并将这些提供给所有人。互联网的创新本质不是来自少量像爱因斯坦那样的人。它来自任何人都可以使用和改进的 RFC，它们由成千上万聪明人，而不是唯一的聪明人研制而成。它来自于任何人都可以使用和改进的开源软件。它来源于共享、社区的规模，良好的解决方案的持续增值和对不好的解决方案的废弃。

因此，下面是创新的另一种理论：

1. 有一个无限大的问题 / 解决方案的范围。
2. 根据外部条件且随着时间的推移，这个范围会发生变化。
3. 我们只能准确感知我们接近的问题。
4. 我们可以利用解决方案市场来对问题的经济成本 / 效益排序。
5. 有针对任何可解问题的最优解。
6. 我们可以试探性地和机械地接近这个最佳的解决方案。
7. 我们的智能可以让这个过程更快，但不会取代它。

对这个理论有下面几个推论：

- 个人创造力不如过程重要。聪明的人可能工作得更快，但他们可能会工作在错误的方向上。这是现实，集体愿景促使我们诚实和团结。
- 如果我们有一个好的过程，那么我们并不需要路线图。当解决方案争夺市场份额时，功能将会随着时间推移而出现。
- 我们发明解决方案的数量不如发现它们那么多。完全同情创意的灵魂，它只是一台喜欢擦亮自己的自我和收集因果报应的信息处理机器。
- 智能是一个社会效应，虽然它给人的感觉是个人的。一个与别人隔离的人最终会停止思考。离开他人的帮助，我们既不能收集问题，也不能测量解决方案。
- 社区的规模和多样性是一个关键因素。更庞大、更多元化的社区会收集更多的相关问题、更准确地解决这些问题，并且会比一小队专家更快地做到这一点。

359 所以，当我们相信孤立的专家时，我们就犯了经典的错误：专注于创意，而不是问题；专注于错误的问题；对解决问题的价值判断失误；不使用自己的作品，和对实际市场的其他许多误判。

我们可以把前面的理论转化为一个可重用的过程吗？在 2011 年年底，我开始记录 C4 和类似的合同，并同时在 ØMQ 和闭源项目中都使用它们。基本过程是一种我称之为“简约化的设计”，或简写为 SOD。这是开发简单而优雅的产品的可重复的方式，它将人们组织到灵活的、能够快速而廉价地定位问题领域的供应链中。它们通过构建、测试，并保持或丢弃被称为“补丁程序”最小的可行的解决方案来完成这项工作。活着的产品包括一长串补丁程序，它们被依次相叠地应用。

SOD 首先是相关的，因为它就是我们演进 ØMQ 的方法。这也是我们将在第 7 章开发大规模的 ØMQ 应用程序中使用的基础设计过程。当然，你可以对 ØMQ 使用任何软件架构方法论。

为了更好地了解我们最终如何使用 SOD，让我们先来看看它的各种替代品。

垃圾桶化的设计

大型企业中最流行的设计过程似乎是垃圾桶化的设计，或简称 TOD。TOD 灌输下面的信仰：为了赚钱我们只需要有伟大的创意。这是顽固的废话，但对缺乏想象力的人却是一根强大的拐杖。该理论说的创意是罕见的，所以关键是要捕捉它们。这就像非音乐家被一个吉他手镇住了，却没有意识到，伟大的人才是如此廉价，他最终会为硬币在街头卖艺。

TOD 的主要输出是昂贵的“构思”：概念、设计文档，以及直接进入垃圾桶的产品。它的工作原理如下：

- 创意人拿出一长串“我们可以做 X 和 Y”的清单。我曾见过将一个产品可以做的所有令人惊奇的事情都列举出来的详细清单。我们都犯过这个错误。当然，一旦构思生成的创造性工作已经发生，剩下的只是执行的问题。
- 因此，管理者和他们的顾问将他们的光辉思想传递给负责创建大量珍贵精致的设计文档的设计师。设计师拿到思路管理者想出的几十个创意，并把它们变成数百个改变世界的设计。
- 这些设计被交给工程师，他们摸不着头脑，不知道谁想出了这样的废话。他们开始反驳，但设计来自高层，而且真的，与创意人和价格昂贵的顾问争论，这不是工程师胜任的事。
- 因此，工程师悄悄回到自己的小隔间，被侮辱和威胁进入到构建巨大但“哦，好优雅” 360
的垃圾堆中。这是极其劳累的工作，因为设计没有考虑实际成本。轻微的率性可能需要数周的工作来构建。当项目被延迟时，管理者就威逼工程师在晚上和周末加班工作。
- 最终，貌似能工作的某个东西出了门。这是摇摇欲坠、脆弱、复杂和丑陋的。设计师诅咒工程师的无能并聘用更多的顾问给猪涂口红（译者注：意指为了欺骗或者诱惑他人而把某件事物粉饰得更有吸引力，但实际上是换汤不换药，类似“别以为穿了马甲，我就不认得你”），慢慢的，产品开始显得好一点了。
- 这时候，管理者已经开始尝试销售产品，他们震惊地发现，没有人想要买它。他们不屈不挠地、勇敢地打造百万美元的网站和广告活动，向公众解释为什么他们绝对需要这种产品。他们经营其他业务，以在市场上推销该产品。
- 经过 12 个月激烈的市场营销，该产品仍然没有创造利润。更糟的是，它遭受到巨大的失败并作为一场灾难在新闻界烙下了骂名。该公司悄悄把它下架，解雇顾问，从一个小的初创公司购买了一款竞争产品，并把它贴牌为自己的版本 2，数亿美元投资最终打了水漂。
- 同时，另一个有远见的经理在组织的某处与一些营销人多喝了点龙舌兰酒，并拥有了另一个绝妙的创意。

如果不是那么常见，垃圾桶化的设计将是一个讽刺。大型企业构建的准备推向市场的 20 个产品中大约有 19 个是失败的（是的，87%的统计数据都是现场编造的）。20 个中剩下的 1 个可能成功，因为竞争对手是如此糟糕，而营销又是如此咄咄逼人。

TOD 的主要教训相当简单，但难以接受。它们是：

- 创意不值钱。没有例外。不存在绝妙的创意。任何试图开始讨论“哦，我们可以做到这一点！”的人都应该被人用为旅行传教士储备的所有激情来打倒。这就像坐在山脚下的一个咖啡厅中，喝着热巧克力，并告诉别人：“嗨，我有一个好主意，我们可以爬上那座山！并在山顶建造小屋！两间桑拿浴室！和一个花园！嘿，我们可以让它采用太阳能供电！老兄，那是真棒！我们应该把它漆成什么颜色呢？绿色！不，蓝色！好了，你去做它吧，我会留在这儿制作电子表格和图形！”
- 361 良好的设计过程的出发点是收集真实的人们所面临的实际问题。第二步是用下面的基本问题，“解决这个问题值多少钱？”来评估这些问题。做完这件事后，我们就可以收集到值得解决的问题集。
- 实际问题的良好解决方案，将会成功地成为产品。它们的成功将取决于解决方案有多么价廉物美，以及问题有多么重要（可悲的是，还取决于有多大的市场营销预算）。但它们的成功也将取决于利用它们需要付出多少努力，换句话说，它们有多么简单易用。

现在，在杀死完全无关的龙后，我们来攻击复杂性的恶魔。

复杂化的设计

真正不错的工程团队和小型公司通常可以建立像样的产品。但绝大多数产品最终仍因为过于复杂而不如预期那么成功。这是因为即使是最好的专业团队，也往往固执地应用我称之为*复杂化的设计*，或简称 COD 的过程，其工作原理如下：

- 管理层正确识别一些有趣的、具有经济价值且困难的问题。在这样做时，他们已经超越了任何 TOD 的团队。
- 团队以饱满的热情，开始构建原型和核心层。这些原型和核心层如设计的那样工作，因此是鼓舞人心的，团队停下来进入紧张的设计和架构的讨论，最终得出看起来美丽和优雅的坚实架构。
- 管理层回来并用更加困难的问题挑战他的队伍。我们倾向于物有所值，因此在他们的头脑中，问题解决起来越困难和昂贵，解决方案就应该越有价值。
- 团队，作为工程师，因此热爱不断地构建东西。他们构建、构建、再构建，并最终构建出大规模的，完美设计的复杂度。
- 产品进入市场，而市场挠着它的头，并问道：“说真的，这是你可以做到的最好的东西吗？”人们确实会使用这个产品，特别是如果他们在攀登学习曲线时不花自己的钱的时候。
- 管理层会从它的大客户获得积极反馈，这些大客户共享同样的高成本（在培训和使用方面）意味着高价值的想法，因而将继续推动这个过程。
- 同时在全球的某处，一支小团队使用一个更好的过程解决了同样的问题，并在一年

后将这个市场砸成了碎片。

COD 的特点是一支团队痴迷于以集体妄想的形式解决错误的问题。COD 的产品往往是庞大、有事业心、复杂的，并且不受欢迎。很多开源软件都是 COD 过程的产物。让工程师们停止用扩展设计来覆盖更多的潜在问题，这是极其艰难的。他们认为，“如果有人想要做 X 会怎样？”，但从来没有问自己：“解决 X 的真正价值是什么？”

COD 在实践中的一个很好的例子是蓝牙，一个复杂的、过度设计的一组令用户讨厌的协 362
议。它继续存在只是因为在一个大规模的专利产业中没有真正的替代品。蓝牙技术是完全安全的，但这对于一个邻近协议来说几乎是无所谓的。与此同时，它缺乏供开发者使用的一个标准 API，这意味着在应用程序中使用蓝牙真的是很昂贵的。

在 *#zeromq* IRC 频道，Wintre 曾经写道，很多年前，当他“发现 XMMS 2 有一个能够工作的插件系统，但不能真正演奏音乐”时，他是多么恼火。

COD 是一种大型“兔子入座”的形式，在其中的设计师和工程师无法将他们自己从其工作的技术细节中撇清。他们增加越来越多的功能，完全误读了他们的工作的经济性。

COD 的主要经验教训也很简单，但对专家来说很难接受。它们分别是：

- 制作你不急需的东西是毫无意义的。无论你多么有才华，或杰出，如果你只是坐下来制作人们实际上并没有需求的东西，那么你最有可能是在浪费你的时间。
- 问题是不均等的。有些很简单，有些则很复杂。讽刺的是，解决简单的问题往往比解决真的很难的问题对更多的人有更多的价值。如果你允许工程师只是对随机的东西开展工作，他们大多会集中于最有趣的，但最不值得做的事情上。
- 工程师和设计师喜欢做东西和装饰，而这不可避免地导致复杂性。关键是有一套“停止机制”：一种来设置短时间、硬性的最后期限的方法，它迫使人们只对最关键的问题做出更小、更简单的解决方案。

简约化的设计

最后，我们终于到达了罕见的，但珍贵的简约化的设计，或简称 SOD。这个过程开始于一种认识：除非我们开始制作它，否则我们不知道我们必须做什么。想出创意或大型设计不只是浪费，它是设计真正准确的解决方案的一个直接障碍。真正内涵丰富的问题隐藏得像一座遥远的山谷，而除主动侦察外的任何活动都是创建一个隐藏那些遥远的山谷的迷雾。你需要保持移动、轻装，并迅速采取行动。

SOD 的工作原理如下：

- 我们（通过观察人们如何使用技术或其他产品）收集了一组有趣的问题，我们将这些问题按从简单到复杂的顺序进行排队，寻找并确定使用模式。
- 363 我们选取最简单，最引人注目的问题，我们以最小可能的解决方案，或“补丁程序”解决了这个问题。每一个补丁程序都以一种极其精简的方式正好解决一个真正的、商定的问题。
- 我们采用一种衡量补丁程序质量的措施，即“还可以更简单地开发这个补丁程序，同时仍然解决了上述问题吗？”我们可以用为了使用这个补丁程序，用户必须了解或猜测的概念和模式的方式来衡量复杂性。用户要学习的东西越少，补丁程序的质量就越好。一个完美的补丁程序无须用户学习任何东西就解决了一个问题。
- 我们的产品开发是由一个解决了“我们需要一个概念证明”问题的补丁程序组成，然后再连绵不绝地演进，通过堆积在彼此顶部的成千上万个补丁程序，得到一个成熟的系列产品。
- 除了一个补丁程序，我们不会做任何其他东西。我们使用要求每一个活动或任务都关系到真正和商定的问题，并明确阐述和记录的正式流程来实行这项规则。
- 我们把我们的项目构建到供应链中，其中每个项目都可以提供问题给它的“供应商”，并获得补丁程序的回报。供应链产生了“停止机制”，这是因为当人们不耐烦地等待一个答案时，我们必须缩短我们的工作。
- 个人可以自由地在任何项目上工作，并在他们觉得值得的任何一个地方提供补丁程序。没有人“拥有”任何项目，除强制执行正式流程以外。一个单独的项目可以有很多变种，每一个都是相互竞争的不同修补程序的集合。
- 项目从正式并得到记录的界面导出，使得上游（客户端）的项目不知道发生在供应商项目的变化。因此，多个供应商的项目都可以争取客户端项目，从而创建一个自由竞争的市场。
- 我们把我们的供应链联系到真正的用户和外部客户，而利用快速迭代驱动整个过程，这使得从外部用户收到的问题在几个小时内得到分析、评价，并通过补丁程序来解决。
- 在每一个时刻，从最初的补丁程序开始，我们的产品都是可交付的。这是必不可少的，因为补丁程序有一大部分是错误的（10% ~ 30%），只有通过向用户提供产品，我们才能知道哪个补丁程序本身已成为需要解决的问题。

SOD 是一种*爬山算法*，在一个未知的领域寻找最显著问题的最佳解决方案的一种可靠方法。为了成功地使用 SOD，你不需要成为一个天才，而只需要能够看到活动的迷雾和通向新的实际问题的过程之间的差异。

364 人们已经指出，爬山算法具有已知的限制。主要是人们被局部峰值卡住。但是，这仍然是事物本身的发展规律：在很长一段时间中收集微小的增量改进。这里没有聪明的设计师。我们通过广泛铺开整个领域来降低局部峰值的风险，但它是有点实际意义的。限制

不可选择，它们是物理定律。该理论认为，这就是创新真正起作用的方式，所以最好去拥抱它、使用它，而不是去尝试在神奇思维的基础上开展工作。

而事实上，一旦你将所有的创新看作为或多或少成功的爬山，你就知道为什么有的团队、公司和产品会卡在一个前景递减的乌有之乡了。他们根本不具备多样性和集体智慧来找到更好的山来爬。当诺基亚干掉它的开源项目时，它要了自己的命。

一个真正的好设计师和一支优秀的团队可以使用 SOD 快速准确地打造世界一流的产品。为了从 SOD 获得最大的好处，设计师必须从第一天开始就连续使用该产品，并发展它嗅出问题的能力，这些问题包括不一致、令人惊讶的行为，以及其他形式的摩擦。当然，我们也忽略了很多烦恼，但一个好的设计师会将这些挑选出来并想着如何修补它们。设计就是为了去除在使用产品过程中的摩擦。

在一个开源的环境中，我们公开地做这项工作。不存在“让我们打开这段代码”的揭晓时刻。如果有的项目没有做到公开，我认为它们迷失了开源的重点，这个重点是你探索与你的用户打交道，并在架构的种子周围建立社区。

职业倦怠

ØMQ 社区一直并且仍然在很大程度上依赖于个人无偿的努力。我这样想，每个人都在以某种方式为他们的贡献得到补偿，而且我相信，为 ØMQ 做贡献意味着在一种非常有价值的技术上获得经验，这将促进专业上的选择。

然而，并非所有项目都那么幸运，如果你使用开源软件工作或你的工作就是开发开源软件，你应该明白志愿者面对的职业倦怠的危险。这适用于所有的公益社区。在本节中，我将解释都有哪些原因会导致职业倦怠，如何认识它，如何防止它，并且（如果发生）如何尽量面对它。免责声明：我不是心理医生，这部分是基于我在过去的 20 年自己在公益环境，包括在自由软件项目和非政府组织，如在 FFII 中的工作经验。

在公益方面，我们预期自己的工作没有直接或明显的经济激励。也就是说，我们牺牲家庭生活、职业发展、空闲时间与健康，以完成我们已经决定要完成的一些目标。在任何项目中，我们都需要某种形式的奖励，使每一天值得持续下去。大多数公益项目的回报 365
非常间接，表面上是完全“不经济”的。大多数情况下，我们做事情是因为人们说，“嘿，太棒了！”付出就有收获是一个强大的动力。

但是，我们是经济性的生物，并且，如果一个项目花费了我们大量的工作，而却没有某种（金钱、名誉、一份新工作……）经济回报，我们迟早会开始受苦。在某一阶段，似乎我们的潜意识只是变得厌烦，说：“我受够了！”不肯再继续下去。如果试图强迫自己，

我们可能最终会生病。

这就是我所谓的“职业倦怠”，虽然这个词也用于其他类型的疲惫。在经济奖励过少的项目中投资过大，时间太久就会产生倦怠。我们非常善于操纵自己和他人，而这往往是引起职业倦怠过程的一部分。我们告诉自己，这是一个良好的事业，而其他人做得不错，所以我们也应该做得很好。

当我在类似 Xitami 的开源项目中感到职业倦怠时，我清楚地记得我的感受。我干脆停止了在该项目上的工作，拒绝答复任何更多的电子邮件，并告诉人们忘掉它。你可以辨认出一个职业倦怠的人。他们变得消沉，而每个人都开始说，“他举止怪异、郁闷，或者累了……”

诊断方法很简单。有受害者在不以任何方式回报的项目上干了很多工作吗？他们做出了特殊牺牲吗？他们丢掉或放弃在项目上的工作或研究工作吗？如果你回答“是”，那这就是职业倦怠。

下面是我已经研究了多年的三个简单原则，它们可以减少我所工作的团队的职业倦怠的风险：

- *没有人是不可替代的*。无论在关键的还是流行的项目中做公益工作，一个不能为自己的工作设限的人身上的责任集中度可能是造成职业倦怠的主要因素。这是一个管理的真理：如果在你的组织中有人是不可替代的，那么请摆脱他或她。
- *我们需要靠日常工作来支付账单*。这可能很难，但似乎很有必要。从别的地方挣钱可以更容易地维持一个要做出牺牲的项目。
- *必须告诫人们有关职业倦怠的问题*。这应该是高校的一门基本课程，因为公益工作要成为年轻人专业地尝试的更常见方式。

当一个人在一个重要的项目中孤独地工作时，你就知道他的保险丝熔断是迟早的事。这实际上是完全可以预测的：它会在大约 18 ~ 36 个月中发生，这取决于他们个人和他们在私人生活中面临多大的经济压力。在一个吃力不讨好的项目中，我从来没见过有任何人在半年后就耗尽精力，也没有见过能持续工作五年的。

366 治愈倦怠有一个简单的，至少在某些情况下有效的办法：得到体面的工作报酬。然而，这个办法却几乎摧毁志愿者享受的移动自由（即跨越无限的问题领域）。

成功模式

我将用在软件工程中的一系列的成功模式来结束这一不含代码的章节。这些模式的目标是找出某些造就辉煌成功而不是悲惨失败的精髓。它们被某个经理描述为“疯狂的教条”，

又在同一天被一位同事描述为“其他任何东西都是该死的精神病”。对我来说，它们是科学。但请将懒惰的完美主义者和其他模式作为工具来使用，锐化它，并且如果找到其他更好的东西，就将其扔掉。

懒惰的完美主义者

如果某种东西不是我们可以识别并且必须解决的一个问题的一个精确的、最小的解决方案，那就永远不要去设计它。

懒惰的完美主义者花费了大量的空闲时间观察别人并识别值得解决的问题。他寻找就这些问题达成的协议，总是问，“什么是真正的问题？”然后他准确而微创地转去建立（或让别人建立）一个问题的可用的解决方案。他自己使用，或让他人使用这些解决方案。他重复此操作，直到所有的问题都已解决，或时间或金钱耗尽。

仁慈暴君

治理大军团与治理小部队的原则是一样的：仅仅是拆分他们的人数的问题。——孙子兵法（译者注：原文为“凡治众如治寡，分数是也。”）

仁慈暴君将大问题拆分成较小的问题，并将它们丢给重点关注某个问题的小组。他以API和我们将在下一章中读到的“无协议”的形式，代理这些小组之间的合同。仁慈暴君构造一个供应链，它从问题开始，并产生可用的解决方案。他对供应链如何工作毫不关心，并且没有告诉人们去做什么，或如何做好自己的工作。

天和地

理想的团队由两方组成：一方编写代码，而另一方提供反馈。

天和地一起作为一个整体，虽然近在咫尺，但它们通过议题跟踪器正式沟通。天从他人和个人使用的产品中找出问题，并把这些问题反馈给地。地迅速地用可测试的解决方案来解答。天和地可以在一天内完成几十个议题。天与其他用户交流，而地与其他开发者交流。天和地可能是两个人，或者两个小团队。

门户开放 367

知识的精确性来自多样性。

门户开放接受几乎任何人的贡献。她不争论质量或方向，而是让别人争论并变得更加投入。她认为，即使是一个巨魔也将为团队带来更多的不同意见。她让该团队形成了关于什么进入稳定的代码的意见，她在仁慈的暴君的帮助下强制实施此意见。

大笑的小丑

完美性排除参与者。

大笑的小丑，经常充当快乐的败将，他不提出高竞争力的要求。相反，他的滑稽动作和装模作样的企图招惹别人进入他自己的悲剧来抢救他。然而，不知怎的，他始终确定需要解决的正确问题。人们都在忙着证明他是错的，他们没有意识到他们正在做有价值的工作。

留心的将军

不是制订计划，而是设定目标、制订战略和战术。

留心的将军在未知的领域工作，解决那些被隐藏，直到它们迫近才凸显的问题。因此，他不制订计划，但一直在寻找机会，然后迅速而准确地利用机会。他在该领域制订战术和战略，并把这些教导给他的手下，使他们既能够独立地移动，也能在一起工作。

社会工程师

知己知彼，百战不殆。——孙子兵法

社会工程师读取与她一起工作和为之工作的人的芳心。她问大家，"是什么让这个人生气、没有安全感、议论、平静、幸福呢？"她研究他们的情绪和性格。有了这些知识，她可以鼓励那些有用的人，并阻止那些没用的人。社会工程师从不表现出她自己的情绪。

不朽的园丁

上下一心的军队将赢得胜利。——孙子兵法（译者注：原文为"上下同欲者胜。"）

当越来越多的人进入项目时，不朽的园丁从一粒小种子开始，逐步地培养一个过程。他为确切的原因，征得每个人的同意来制作每一个更改。他从来没有高高在上地规定一个过程，而是让别人来达成共识，然后他强制执行该共识。以这种方式，每个人都共同拥有这个过程，而且他们通过拥有这个过程，自己紧密地依附于这个过程。

368

滚石

过河之后，你应该尽量远离它。——孙子兵法（译者注：原文为"绝水必远水。"）

滚石接受自己的死亡和无常。她没有依恋她过去的工作。她承认我们制作的这一切的目的地是垃圾桶，这只是一个时间问题。利用精确、最小的投资，她能迅速摆脱过去，专注于现在和不久的将来。最重要的是，她没有自我，没有因别人的行为而受到伤害的自豪感。

海盗帮

像所有的知识一样，代码在作为集体而不是私人财产时效果最好。

海盗帮会自由组织周围的问题。他接受权威，只要权威提供目标和资源。海盗帮会拥有和共享他制作的所有东西：每一件工作都由在海盗帮中的其他人完全重新混合。当新的问题出现时，该团伙正迅速转移，而他很快就抛弃旧的解决方案，如果这些旧方案阻碍团结。无任何人士或团体可以垄断供应链的任何部分。

快闪族

水根据其流向地面的形状塑造它的路线。——孙子兵法（译者注：原文“水因地而制流，兵因敌而制胜。”）

快闪族在需要的空间和时间走到一起，然后尽其所能地快速分散。物理上相互接近对高频的沟通是必不可少的，但随着时间的推移，它会产生技术壁垒，导致天与地的分隔。快闪族往往在各地出差时收集了大量的飞行常旅客里程。

加那利看守

痛苦一般不是一个好兆头。

加那利看守用他自己的疼痛程度和所观察到的那些与他一起工作的人的疼痛程度来衡量一个组织的质量。他把新的参与者带入现有的组织，使他们能够表达无辜的原始痛苦。他可以用酒精来让别人用语言表达他们的痛点。他问别人和自己，“你在这个过程中快乐吗，如果不快乐，为什么会这样呢？”当一个组织引起自己或他人的疼痛，他将其作为一个待解决的问题。人们应该在工作中感到快乐。

执行绞刑的刽子手 369

永远不要在别人犯错误时打断他们。

执行绞刑的刽子手都知道，我们只会从失误中学习，他给别人丰富的绳索用来学习。当到时间时，他只轻轻地拉动绳子。微小的拖动提醒其他人的位置岌岌可危。允许他人从失败中吸取教训，给人留下来的好理由，和离开的坏借口。执行绞刑的人的耐心是无止境的，因为学习的过程没有捷径。

历史学家

保持公共记录可能是乏味的，但它是防止串通的唯一途径。

历史学家迫使讨论进入公众视野，以防止在自己的工作领域中串通。海盗帮取决于充分和平等的沟通，不依赖于一时的存在。没有人真的会读取档案，但简单的可能性会阻止

大部分的滥用。历史学家鼓励为工作采用正确的工具：电子邮件用于瞬态讨论、IRC 用于喋喋不休、维基用于知识、议题跟踪器用于记录各种机会。

煽动者

当一个人知道自己将在两周后接受绞刑时，会令人惊奇地集中他的心思。

——Samuel Johnson

煽动者造就最后期限、敌人，和偶尔的不可能性。团队在他们没有时间说废话时运作得最好。最后期限将人们团结在一起，集中集体的思想。外部的敌人会促使被动的团队行动。煽动者绝不会太认真看待最后期限。产品随时准备出货。但她轻轻地提醒赌注的球队：倘若失败，我们都会去找工作。

神秘人

当人们争论或抱怨时，只需给他们写一段孙子的语录。——Mikko Koppanen

神秘人从不直接争论。他知道，与一个情绪化的人争论只会产生更多的情绪。相反，他回避了讨论。很难对一位中国将军生气，特别是当他已经死了 2400 年。当人们坚持把事情搞糟时，神秘人扮演刽子手的角色。

第 7 章 371

使用ØMQ的高级架构

大规模使用 ØMQ 的影响之一是，因为我们可以比以前快许多地构建分布式架构，所以我们的软件工程过程的局限性变得更明显。慢动作中的错误往往更难看到（或者说，更容易理顺了）。

我对 ØMQ 工程师团队的教学经验表明，仅对他们解释 ØMQ 的工作原理，对于随后期待他们开始构建成功的产品是很不够的。与消除摩擦的任何技术一样，ØMQ 为大失误打开了大门。如果 ØMQ 是分布式软件开发的 ACME 火箭推进式跑鞋，我们很多人，比如 Wile E. Coyote，都会“砰”地一声全速进入众所周知的沙漠悬崖。

在第 6 章我们看到 ØMQ 本身使用一个正式的过程来应对更改。我们多年来建立这个过程的一个原因，是为了在库本身的开发中防止再次发生类似冲向悬崖的情况。

部分是为了减缓移动速度，部分是要确保当你快速移动时，你去往*正确的方向*，这是必须的，亲爱的读者。这是我的标准面试哑谜：任何软件系统中最稀缺的财产，绝对最难做对的事是什么，缺少了它们就会导致绝大多数项目或慢或快地死亡呢？答案不是代码的质量、资金、性能，甚至也不是普及（虽然这是一个接近的答案）。答案是*准确性*。

准确性是我们面临的一半挑战，并适用于任何工程性工作。另一半挑战则是具体的分布式计算本身，这规定了，如果我们要建立大型架构，需要解决的一系列问题。我们需要对数据进行编码和解码，我们需要定义协议来连接客户端和服务器，我们需要确保这些协议能抵御攻击者，而且我们需要制造健壮的软件栈。要知道，异步消息传递是很难做对的。

本章将应对这些挑战，首先是对如何设计和构建软件的基本重估，并以一个用于大型文 372
件分发的分布式应用程序的形式的完整例子来结束。

我们将讨论以下内容丰富的主题：

- 如何安全地从创意过渡到能工作的原型（MOPED 模式）
- 将你的数据作为 ØMQ 消息序列化的不同方式
- 如何用代码生成二进制序列化的编解码器
- 如何使用 GSL 工具来建立自定义的代码生成器
- 如何撰写和许可一个协议规范
- 如何在 ØMQ 上进行可快速重新启动的文件传输
- 如何实现基于信用的流量控制
- 如何将协议的服务器和客户端构建为状态机
- 如何制作一个在 ØMQ 之上的安全协议
- 一个大型的文件发布系统（FileMQ）

用于弹性设计的面向消息模式

在本节中，我将介绍用于弹性设计的面向消息模式（MOPED），一个用于 ØMQ 架构的软件工程的模式。我曾经在“MOPED”和“BIKE”（Backronym-Induced Kinetic Effect，诱导动力的影响的首字母缩写）两个缩写词间选择。后者是“BICICLE”的简称（Backronym-Inflated See if I Care Less Effect），它是“得意地看看我是否不在乎影响”的首字母缩写。在生活中，一个人要学会选用最少尴尬的选项（译者注：单词 BIKE 和 BICYCLE 都表示自行车，作者在此暗示 BIKE 这个缩写词尴尬）。

如果你一直仔细地阅读这本书，你就已经在实践中看到 MOPED 了。第 4 章的管家模式的开发是它的一个近乎完美的案例。但可爱的名字胜过千言万语。

MOPED 的目标是定义一个过程，通过它我们可以取得针对一个新的分布式应用程序的粗略用例，并在不到一个星期的时间内用任何语言从“Hello World”编写出全面工作的原型。

使用 MOPED，你重新培育而不只是构建失败风险最小的能工作的 ØMQ 架构。通过专注于合同，而不是实现，你就避免了过早优化的风险。通过超短的基于测试的迭代来驱动设计过程，你可以在添加更多功能之前对已有什么作品更加肯定。

我们可以将其分为 5 个具体步骤，如下所示：

1. 内部化 ØMQ 语义。
373 2. 描绘一个粗略的架构。
3. 决定合同。

4. 编写一个最小的端到端解决方案。
5. 解决一个问题，然后重复。

第 1 步：内部化的语义

你必须学习和消化 ØMQ 的“语言”，即，套接字模式以及它们的工作方式。学习一门语言的唯一方法是使用它。这种投资没有办法避免，没有在你睡觉时可以听的磁带，也没有插上就可以神奇地变得更聪明的芯片。阅读本书的第 1 部分，完成代码示例，明白这是怎么回事，以及（最重要的）自己编写一些例子，然后把它们扔掉。

在某一特定时刻，你会感觉到在你的大脑中发出“咔哒”一声。也许是一个奇怪的辣椒引起的梦，在梦中，小 ØMQ 任务跑来跑去试图活吃了你。也许你只是觉得，“啊哈，那么这就是它的含义！”如果我们正确地做了我们的工作，这应该需要两到三天。无论这用了多久，除非你开始用 ØMQ 套接字和模式的方式思考，否则你就还没有为第 2 步做好准备。

第 2 步：描绘一个粗略的架构

以我的经验，能够描绘你的架构的核心是必需的。这不但有助于别人了解你在想什么，而且也可以帮助你思考你的想法。实在是没有比使用白板更好的方式来向你的同事解释你的想法了。

你并不需要把它做对，而且你并不需要让它完整。你需要做的是将你的架构分解成有意义的部件。软件架构（相对于建造桥梁）的好处是，如果你已经隔离了各个层，你真的可以廉价地替换整个层。

首先选择你所要解决的核心问题。忽略任何不必要的问题：你会在以后添加它。这个问题应该是一个端到端的问题：横跨峡谷的绳子。

例如，一个客户要求我们用 ØMQ 制作一个超级计算集群。客户端创建工作包，将其发送到一个代理，由它们分发到工人（在快速的图形处理器上运行），收集反馈结果，并将其返回给客户端。

横跨峡谷的绳子是一个客户端跟一个代理交流，而该代理又与一个工人交流。我们画出
三个方框：客户、代理和工人。我们画出从框到框的箭头，显示单向流动的请求和倒流 374
的响应。这就像我们在前面章节中看到的很多图。

为了简约化，我们的目标不是定义一个真正的架构，而是扔出跨越峡谷的绳子来引导我们的过程。我们会让架构随时间的推移渐趋完整和逼真：例如，添加多个工人、增加客

户端和工人 API、处理故障，等等。

第 3 步：决定合同

一个好的软件架构依赖于合同，并且如果它们越明确，扩展性就越好。你不用关心事情是如何发生的，只关心结果即可。如果我发送一封电子邮件，我不关心它是如何到达目的地的，只要合同得到了尊重（它在几分钟内到达，它没有被修改，而且它不会丢失）。

而为了建立一个工作得很好的大系统，在实现前你必须着眼于合同。这听起来可能很明显，但人们往往忘记或忽略了这一点，或者只是太害羞而不肯把它强加给自己。我希望我能说 ØMQ 已经做好这一点，但多年来我们的公共合同是二流的事后的想法，而不是原发于你面前的工作部件。

那么，什么是分布式系统的合同呢？依据我的经验，存在两种类型的合同：

- 针对客户端应用程序的 API。回顾一下第 6 章中的软件架构的心理因素。API 必须尽量绝对简单、一致和为人熟知。是的，你可以从代码生成 API 文档，但必须首先设计它，而设计一个 API 往往是困难的。
- 连接部件的协议。这听起来像火箭科学，但它实际上只是一个简单的技巧，并且是 ØMQ 特别容易做的一个技巧。事实上，它们编写起来是如此简单，需要那么少的官僚主义，所以我把它们称为“unprotocols”（反协议）。

你将编写最小的合同，它们大多只是放置标记。多数消息和大多数 API 方法都将缺失或为空。你也希望以吞吐量、时延、可靠性等形式写下任何已知的技术要求。这些都是你将接受或拒绝的任何特定工作部件的标准。

第 4 步：编写一个最小的端到端解决方案

我们的目标是尽可能快地测试出整体架构，制作出调用 API 的骨架应用程序，以及实现每个协议两侧的骨架栈。你想尽快获得一个能工作的端到端的“Hello World”程序。你希望当你把代码编写出来时能够对其进行测试，并淘汰蹩脚的假设和你犯下的不可避免的错误。不要冲出去并花半年时间编写一个测试套件！相反，请编写一个最小的骨架系统应用程序，让它使用你仍然假设的 API。

375 如果你从实现它的作者的角度设计一个 API，你就会开始考虑性能、功能、选项等。你会使它比它实际需要的更复杂、更不规则、更令人惊讶。但是，这里是窍门（这是一个廉价的窍门）——而如果你从真正利用它来编写应用程序的程序员的角度设计 API，你就会使用所有对你有利的懒惰和恐惧。

用你可以清楚地解释每一个命令，而没有太多细节描述的这样一种方式，在一个 wiki 或共享文件中写下协议。去掉所有真正的功能，因为它只会产生惯性，使得它更难到处移动东西。你可以随时增加重量。不要花力气定义正式的消息结构：采用 ØMQ 的多部分组帧，用尽可能简单的方式传递最小化的东西。

我们的目标是让最简单的测试案例能够工作，没有任何多余的功能。在要做的事情列表中，砍掉一切可以砍掉的任务。忽略来自同事和老板的呻吟声。我将再次重复这句话：你可以随时添加功能，这是比较容易的。但目标是将整体规模保持在最小值。

第 5 步：解决一个问题，然后重复

你现在处于问题驱动的开发的高兴周期中，在这里你可以开始解决有形问题，而不是增加功能。编写说明清楚问题的议题，并为每个议题提出解决方案。当你设计 API 时，请记住你的命名标准、一致性和行为。用简单的文字写下这些内容有助于保持它们的清晰。

从这里，你对架构和代码的每一处更改都可以通过运行测试案例来验证，如果它不能正常工作，则进行修改，如此往复，直到它可以正常工作为止。

现在，你可以遍历整个周期（根据需要扩展测试案例、修正 API、更新协议、扩展代码），每次选取其中一个问题并单独测试其解决方案。每个周期应该需 10 ~ 30 分钟，以及由于随机混乱造成的偶尔秒杀。

Unprotocol

当这个男人考虑协议时，这个男人想到的是委员会长年累月编写的大量的文档。这个男人想到 IETF、W3C、ISO、Oasis、监管俘获、FRAND 专利授权纠纷 ...... 不久之后，这个男人考虑退休，去玻利维亚北部的一个不错的小农场安享晚年，在那里的山上，唯一另外一种不必要倔强的生灵就是嚼咖啡树的山羊。

现在，我对委员会没有什么个人的反对意见。没用的民间人士需要一个带有最低的繁殖风险的地方享受他们的生活，毕竟，那看上去是完全公平的。但多数委员会协议都有复杂性（那些能够工作的）或垃圾（那些我们不谈论）倾向。这有几个原因。一个原因是 376
利害攸关的金钱数量。更多的资金意味着更多的人想在散文中表达自己的特别的偏见和假设。但第二个原因是缺乏良好的抽象来在其上建立协议。人们试图建立可重复使用的协议抽象，像 BEEP。大多数协议没有坚持这种抽象，而那些坚持了这种抽象的协议，比如 SOAP 和 XMPP，是在事物的复杂的一方。

几十年前，当互联网还是一个年轻而温和的东西的时候，协议曾经是简短而亲切的。它们甚至不是“标准”，而是“征求意见的请求”，这是尽可能谦虚的。自从我们在 1995 年创立 iMatix 以来，找到一种让像我这样的普通人没有委员会开销地编写小型、准确的协议，简短而亲切一直是我的目标之一。

现在，ØMQ 确实似乎提供了鲜活的、成功的协议抽象层，它使用“我们将通过随机传输进行多部分消息传递”的工作方式。因为 ØMQ 默默地处理组帧、连接和路由，所以在 ØMQ 之上编写完整的协议规范非常容易，而我已经在第 4 章和第 5 章展示了如何做到这一点。

大约在 2007 年年中，我创办了数字标准组织来定义产生小型标准、协议和规范的更简单的新方法。为我自己辩护一下，这是一个安静的夏天。当时，我写道（*http://www.digistan.org/spec:1*）：一个新的规范应花费“几分钟解释，几小时进行设计，几日编写，几周来证明，几个月来变得成熟，几年来取代”。

2010 年，我们开始把这样的小规范叫作“unprotocol”，有些人可能会误认为它是一个朦胧的国际组织统治世界的一个卑鄙的计划，但它实际上只是意味着“没有外套的协议。”

合同是艰难的

编写合同，也许是大规模架构中最困难的部分。使用 unprotocol，我们能尽可能多地消除不必要的摩擦。但剩下的仍然是需要解决的很难的一系列问题。好的合同（无论是一个 API、协议或租赁协议）必须简单、明确、技术上可靠，同时易于执行。

如同任何技术技能，它是你必须学习和练习的东西。在 RFC ØMQ 网站（*http://rfc.zeromq.org/*）上有一系列规范，这是值得一读的，并且当你发现自己有需要时，就可以使用它们作为你自己的规范的基础。

我试着总结一下我在编写协议的经验中所学到的东西：

- 从简单的开始，并逐步开发你的需求。不去解决你还没有遇到的问题。
- 377 使用非常明确和一贯的语言。协议往往分解成命令和字段，为这些实体使用清晰而简短的名称。
- 尽量避免发明概念。从现有的规范重用一些东西就可以了。对你的听众使用清楚而明显的术语。
- 不做你不能证明有迫切需要的任何东西。你的规范能解决问题，它不提供功能。要为你确认的每个问题做出最简单可行的解决方案。
- 在你构建它的同时实现你的协议，以便你了解每个选择的技术后果。使用一种使得它很难实现的语言（如 C），而不是一种使它很容易实现的语言（如 Python）。

- 在你构建它的同时测试你的规范。针对一个规范的最好的反馈是，在别人不具备你的头脑中的假设和知识时，试图实现它。
- 快速、一致地交叉测试，用别人的客户端连接你的服务器，反之亦然。
- 准备把它抛出来，并根据需要随时重新启动。要对这一点做出规划，举例来说，通过对架构进行分层以便你可以保留一个 API，但改变底层协议。
- 只使用独立于编程语言和操作系统的结构。
- 分层解决大问题，使得每一层都是独立的规范。谨防创建庞大的整体协议。考虑如何重用每一层。思考不同的团队如何可以在每一层构建竞争性的规范。

最重要的是，把它写下来。代码不是一种规范。有关书面规范的重点是，不管它有多么软弱，它都可以被系统地改善。通过写下一个规范，你将能够发现不一致和灰色地带，这些东西都是不可能在代码中看到的。

如果这听起来很难，不要太担心。使用 ØMQ 的一种不太明显的好处是，它减少了编写一个协议规范大约 90%以上的必要努力，因为它已经处理了组帧、路由、排队，等等。这意味着你可以快速尝试，低成本地犯错，从而迅速学习。

如何编写 Unprotocol

当你开始编写一个 unprotocol 规范文档时，请坚持采用一致的结构，以让你的读者知道会发生什么。下面是我使用的结构。

- 封面部分：带有写在一行中的摘要、规范的 URL、正式的名称、版本、负责人。
- 正文的许可证：对于公共规范是绝对需要的。 378
- 变更过程：即，我作为读者能怎么解决规范中的问题？
- 语言的使用：必须、可以、应该等，具有对 RFC 2119 的引用。
- 成熟度指标：这是一个实验、草稿、稳定、遗产，还是已退休的版本？
- 协议目标：它试图解决什么问题？
- 正式语法：防止由于文本的不同解释导致的争论。
- 技术说明：每个消息的语义、错误处理等。
- 安全讨论：明确的，协议的安全程度。
- 参考文献：对其他文件、协议等的引用。

编写清晰、富有表现力的文本是很难的。请避免试图描述协议的实现。请记住，你正在编写一个合同。以清晰的语言描述了每一方的义务和期望，义务的水平，以及对破坏规则的惩罚。不要试图定义各方如何信守交易的一部分。

以下是有关 unprotocol 的一些关键点：

- 只要你的过程是开放的，你就不需要一个委员会：只是做干净、简单的设计，确保任何人都可以自由地改进它们。
- 如果使用现有的许可证，你就不会有法律的后顾之忧。我为我的公开规范使用 GPLv3，并建议你也这样做。对于内部工作，标准的版权是完美的。
- 形式是有价值的。也就是说，学会写正式的语法，如 ABNF（增广巴科斯 - 诺尔范式），并用它来完全记录你的信息。
- 使用类似Digistan的COSS的一种市场驱动的生命周期过程，以使人们当他们成熟（或不成熟）时，正确地把握你的规范。

为什么使用 GPLv3 的公开规范

你选择的许可证对于公共规范尤为关键。传统上，协议在自定义许可下发布，其中作者拥有自己的文本并禁止派生作品。这听起来很不错（毕竟，谁愿意看到一个协议被派生呢），但它实际上是非常危险的。一个协议委员会容易被俘获，并且如果协议是重要的，有价值的，用于俘获的激励也会增长。

一个重要的协议就像一些野生动物那样，一旦被捕获，它往往会死亡。真正的问题是，没有办法来释放在传统许可证下发布的被俘获的协议。单词"*free*"不只是一个用来形容说话或空气的形容词（译者注：形容词 free 翻译为自由），它也是一个动词，而不按
379 所有者的意愿派生一个项目的权利对于避免俘获是必不可少的。

让我简短地解释一下。假设 iMatix 今天编写了一个非常了不起和流行的协议。我们发布规范，而许多人实现它。这些实现快速、完美，并且就像免费啤酒那样免费。这些实现开始威胁到现有的公司，因为他们生产的昂贵的商业产品的速度较慢，无法与之竞争。所以有一天，这些公司的一些代表来到我们在韩国 Maetang-Dong 的 iMatix 办公室，并提议购买我们的公司。因为我们在寿司和啤酒上花费了大量的资金，所以我们感激地接受了。伴随着邪恶的笑声，协议的新主人停止了对公共版本的改善，关闭了这个规范，并加入有专利的扩展。他们的新产品都支持这些，他们接管了整个市场。

当你贡献一个开源项目时，你真的不希望你的努力工作会被一个封闭源代码的竞争对手用来对付你。这就是为什么 GPL 击败了"更宽容"的 BSD/MIT/X11 许可证的原因。因为这些许可证准许欺骗。这个结论对协议的适用丝毫不亚于对源代码的适用。

当你实现一个 GPLv3 的规范时，当然，你的应用程序是你的，它们可以按你喜欢的任何方式来许可。但你可以肯定两件事情。首先，该规范将永远不会被纳入和扩展为专有的形式。本规范的任何派生形式也必须是 GPLv3 的。第二，曾经实现或使用这个协议的人永远不会对它涵盖的任何东西发动专利攻击。

使用 ABNF

我对编写协议规范的建议是学习和使用正式的语法。比起让别人误解你的意思，再从不可避免的错误假设中恢复，学习和使用正式语法的麻烦要少多了。你的语法的目标是其他人：是工程师，而不是编译器。

我最喜欢的语法是由 RFC 2234（*http://www.ietf.org/rfc/rfc2234.txt*）定义的 ABNF，因为它可能是用于定义双向通信协议的最简单和最广泛使用的正式语言。大多数 IETF（互联网工程任务组）规范都使用 ABNF，这是一个值得合作的好伙伴。

下面，我将给出一个 30 秒编写 ABNF 的速成教程。它可能让你想起正则表达式。你把语法编写成规则。每个规则的形式为“名称 = 元素”。一个元素也可以是其他规则（你在下面定义为另一规则的东西），或预先定义的“终端”（如 CRLF，八位字节），或一个数字。RFC 中列出了所有的终端。要定义可选元素，用“元素 / 元素”的形式。要定义重复，用“*”（请参考 RFC，因为它不直观）。要对元素分组，则使用括号。

我不知道这个扩展是否正确，但我随后给元素加上“C:”和“S:”前缀，用来表示它们来自客户端或服务器。

这里有一个名为 NOM 的 unprotocol 的 ABNF 片段，我们将在本章后面再来看它： 380

```
nom-protocol    = open-peering *use-peering

open-peering    = C:OHAI ( S:OHAI-OK / S:WTF )

use-peering     = C:ICANHAZ
                / S:CHEEZBURGER
                / C:HUGZ S:HUGZ-OK
                / S:HUGZ C:HUGZ-OK
```

其实我已经在商业项目中使用了这些关键字（OHAI，WTF）。它们使开发者傻笑和快乐，它们把管理层搞糊涂了。它们是好初稿，但以后你会把它们扔掉的。

廉价或讨厌的模式

在几十年的写作小型和大型协议的工作中，我学到了一个普遍的经验教训，我把这称为“廉价或讨厌”的模式：你可以经常将工作分为两个方面或层次，并分别解决它们，一些用的是“廉价”的方法，另一些用的是“讨厌”的方法。

使廉价或讨厌的方法能够工作的关键洞察力是要认识到，许多协议都混合了用于控制的小容量的同步交互的部分，以及用于数据的大容量的异步部分。例如，HTTP 有一个同

步交互的对话来验证和获取页面，还有一个异步对话用来注入数据。FTP 实际上将这些分解在两个端口上，一个端口用于控制，而另一个端口用于数据。

不将数据与控制分开的协议设计者往往制作出可怕的协议，因为在两种情况下的权衡几乎完全相反。对于控制是完美的东西对于数据却是糟糕的，而对数据是理想的东西却完全不能用于控制工作。当我们想要在同一时间使高性能具有可扩展性和良好的错误检查的时候，这种情况尤为真实。

让我们用传统的客户端 / 服务器用例来分解这些。客户端连接到服务器并进行身份验证。然后，它会要求得到一些资源。服务器做出回应，然后开始发送数据返回给客户端。最终，客户端断开连接或服务器完成，对话就结束了。

现在，在开始设计这些消息之前，让我们停下来思考，并对控制对话和数据流加以比较：

- 控制对话持续时间短，涉及非常少的信息。数据流可能会持续数小时或数天，并涉及数十亿个消息。
- 381 在控制对话中会发生所有"正常"的错误，例如，未通过身份验证，没有找到，需要付费，审查，等等。数据流过程中所发生的任何错误都是例外（磁盘已满，服务器崩溃）。
- 当我们添加更多的选项或参数等的时候，控制对话里的东西会随着时间的推移而改变。数据流几乎不应随时间而改变，因为一个资源的语义在一段时间内是相当恒定的。
- 控制对话本质上是一个同步的请求 - 应答对话。数据流基本上是单向的异步流。

这些差异是重要的。因此，当我们谈论性能时，它仅适用于数据流。将一次性控制对话设计为快速的，这是病理性的。当我们谈论序列化的成本时，这仅适用于数据流。对控制流进行编码 / 解码的成本可能是巨大的，而在许多情况下，它不会改变任何事情。因此，我们利用廉价的方式对控制编码，并且利用讨厌的方式对数据流编码。

廉价方式本质上是同步、详细、描述性和灵活的。一个廉价的消息充满了可以针对每个应用程序改变的丰富的信息。你的设计目标是让这些信息易于编码和解析，对于实验或增长易于扩展，并且针对变化非常健壮，同时向前和向后兼容。协议的廉价部分看起来像这样 ：

- 它对数据使用简单的自我描述的结构化编码，无论是 XML、JSON、HTTP 式的标题，或者是一些其他的编码。任何编码方式都是好的，只要你的目标语言对它有标准简单的解析器。
- 它采用的是直白的请求 - 应答模型，其中每个请求都有一个成功 / 失败的应答。这使得它容易编写正确的廉价对话客户端和服务器。

- 它不会尝试快速，甚至连轻微的尝试都没有。当你做一些只有一次性或每个会话几次的事情时，性能是无所谓的。

廉价方式的解析器是你从架子上取下来后就将数据丢给它的东西。它不应该崩溃，应该不会出现内存泄漏，应该高度宽容，并且应该是比较易用的。这就行了。

但是，讨厌方式本质上是异步、简洁、沉默和不灵活的。一个讨厌方式的消息中携带几乎从不改变的最精简的信息。你的设计目标是让这些信息得到超快的解析，甚至可能无法扩展和试验。理想的讨厌模式看起来像这样：

- 它对数据使用一个手工优化的二进制布局，其中每个二进制位都是精心设计的。
- 它采用纯异步模型，其中一个或两个对等节点发送数据，而无须确认（或者，如果它们确实使用了确认，那么它们用了偷偷摸摸的异步技术，如基于信用的流量控制）。
- 它不会尝试对人友善，甚至连轻微的尝试都没有。当你正在做每秒执行数百万次的 382
东西时，性能是一切。

讨厌方式的解析器是你手工编写的，它单独和精确地写入或读取二进制位、字节、字和整数。它拒绝任何它不喜欢的东西，根本不执行内存分配，永不崩溃。

廉价或讨厌不是一个普遍的模式，不是所有的协议都具有这种二分法。此外，你如何使用廉价或讨厌的方式将取决于你的具体情况。在某些情况下，它可以是一个单独的协议的两个部分。在其他情况下，它可以是一个层叠在另一个之上的两个协议。

错误处理

使用廉价方式或讨厌方式做错误处理相当简单。它具有两种命令和两种引发错误信号的办法：

同步控制命令

错误是正常的：每个请求都有一个响应，要么是 OK，要么是错误响应。

异步数据命令

错误是异常的：坏的命令要么被悄悄丢弃，要么导致整个连接被关闭。

区分几种错误通常是很好的，但要始终使它保持最精简，并只添加你所需要的东西。

序列化数据

当我们开始设计一个协议时，面临的第一个问题就是如何对线路上的数据进行编码。没有通用的答案。序列化数据有半打不同的方法，每种方法都各有利弊。我们将探讨其中的一些方法。

ØMQ 组帧

用于 ØMQ 应用程序的最简单、最广泛使用的序列化格式是 ØMQ 自己的多部分组帧。例如，下面是管家协议（*http://rfc.zeromq.org/spec:7*）定义请求的方式：

第 0 帧：空帧
第 1 帧："MDPW01"（6 个字节，相当于 MDP/ 工人 V0.1）
第 2 帧：0X02（一个字节，相当于 REQUEST）
第 3 帧：客户端地址（封包栈）
第 4 帧：空（零字节，封包分隔符）
第 5 帧以上：请求正文（不透明的二进制码）

383 在代码中读取和写入这些是很容易的。但是，这是控制流的一个经典例子（整个 MDP 是一个经典的例子，真的，因为它是一个同步交互的请求 - 应答协议）。当我们改善 MDP 的第二个版本时，必须改变这种组帧。好极了，我们破坏了所有现有的实现！

向后兼容性是很难的，但将 ØMQ 组帧用于控制流并没有好处。下面是如果我遵循我自己的意见，应该怎么设计这个协议（我将在下一版本修正它）。它分解成一个廉价的部分和一个讨厌的部分，并且它使用 ØMQ 组帧来区分这些：

第 0 帧："MDP/2.0" 的协议名称和版本
第 1 帧：命令标头
第 2 帧：命令正文

我们期望在各种中介（客户端 API、代理以及工人 API）中解析命令标头，并将命令正文体原封不动地从一个应用程序传递到另一个应用程序。

序列化语言

各种序列化语言都有自己的风格。XML 在流行的时候曾经是大的，它在被过度设计时变大，然后陷入了“企业信息架构师”之手，从那以后它就不再有活力了。今天的 XML 是“小型、优雅的语言试图逃脱的某处困境”的缩影。

尽管如此，XML 仍是比它的前辈更好的方法，这些前辈包括如怪物的标准通用标记语言（SGML），而后者比起诸如 EDIFACT 之类的令人折磨的野兽来又是清爽的凉风。因此，序列化语言的历史似乎是一个逐渐显现理智的过程，它隐藏在一大波的反抗 EIA（美国电子工业协会）之类组织尽全力保住其工作成果的斗争后面。

作为一种把数据丢到线路上，并把它找回来的快速而肮脏的“我宁愿辞职也不在这里使用 XML”的方式，JSON 从 JavaScript 的世界杀出来。JSON 只是偷偷摸摸地表达得像 JavaScript 源代码的最精简的 XML。

下面是在一个廉价方式的协议中使用 JSON 的简单例子：

```
"protocol": {
    "name": "MTL",
    "version": 1
},
"virtual-host": "test-env"
```

同样的例子在 XML 中是这样的（XML 迫使我们去创建一个单独的顶层实体）：

```
<command>
    <protocol name = "MTL" version = "1" />
    <virtual-host>test-env</virtual-host>
</command>
```

而采用普通旧式的 HTTP 式的标头如下： 384

```
Protocol: MTL/1.0
Virtual-host: test-env
```

这些都是等价的，只要你不针对验证解析器、模式，和其他“相信我们，这一切都是为了你自己好”的废话走极端。一种廉价方式的序列化语言为你提供了自由实验的空间（忽略任何你不认识的元素 / 属性 / 标头），并且容易编写通用的解析器，例如，将一个命令转换到一个散列表，或反向转换。

但是，这不是完美的。虽然现代的脚本语言能足够轻松地支持 JSON 和 XML，但旧的语言不能。如果使用 XML 或 JSON，你就产生了不寻常的依赖。这也是一个有点像在 C 语言中处理树形结构数据时的痛苦。

所以，你可以根据你的目标语言驱动你的选择。如果你的环境是一种脚本语言，那么去用 JSON。如果你的目标是建立更广泛的系统中使用的协议，那就让事情对 C 语言开发者保持简单，并坚持采用 HTTP 风格的标头。

序列化库

msgpack.org 网站对 MessagePack 序列化库做了如下的介绍：

> 这就像 JSON，既快速又小巧。MessagePack 是一种高效的二进制序列化格式。它允许你将数据在类似 JSON 的多种语言之间交换，但它的速度更快、更小巧。例如，小整数（如标志或错误代码）被编码成一个字节，而典型的短字符串除了字符串本身只需要一个额外的字节。

我打算做也许是不受欢迎的评论，“快、小”是解决非问题的功能。据我所知，该序列库解决的唯一真正的问题是，满足记录消息合同、将实际数据序列化到线路上和把数据从线路反序列化回来的需要。

让我们先从“快、小”的说法开始。它是基于一个两部分的说法:第一，使你的信息更小，并且降低编码和解码的 CPU 成本将对你的应用程序的性能产生明显差异。第二，这对所有消息将是一刀切地同样有效的。

但大多数实际应用程序一般分为两类：要么相比其他成本，如数据库访问或应用程序代码的性能，序列化无论是速度还是编码大小都是可忽略的，要么网络性能确实是至关重要的，因而所有的显著成本都发生在一些特定的消息类型中。

因此，一刀切地瞄准“快速、小巧”是假的优化。你得到的既不是用于你不太频繁的控
385 制流的廉价方式的简单灵活性，也不是用于你的大容量数据流的讨厌方式的极致效率。更糟的是，假设所有消息都是平等地有可能在某些方面破坏你的协议设计。廉价或讨厌方式不仅与序列化策略相关，它也与同步和异步、错误处理和更改的成本相关。

我的经验是，基于消息的应用程序的大多数性能问题可以通过（a）改善应用程序本身及(b)手工优化大容量数据流来解决。而为了手工优化最关键的数据流,你需要欺骗和了解,并利用有关数据的事实，这是一件通用序列化器无法做到的事。

现在让我们讨论文档：明确并正式写入我们的合同，而不仅是在代码中体现的需求。这是一个待解决的有效问题：事实上，如果我们要建立一个持久的、大规模的、基于消息的架构，它是一个主要的问题。

下面是我们使用 MessagePack 接口定义语言（IDL）来描述一个典型消息的方法：

```
message Person {
  1: string surname
  2: string firstname
  3: optional string email
}
```

现在，下面是一个使用谷歌的协议缓冲区 IDL 描述的相同消息：

```
message Person {
  required string surname = 1;
  required string firstname = 2;
  optional string email = 3;
}
```

它能够工作,但在大多数实际情况中,比你通过手写或机械方法产生（我们会得出这样的）

的体面的规范支持的序列化语言强不了多少。你将付出的代价是一个额外的依赖，并很可能比你使用廉价或讨厌方式来做的整体性能更差。

手写的二进制序列化

正如你将从这本书中了解到的，我的系统编程的首选语言是C（升级为C99，具有构造函数/析构函数API模型和通用的容器）。我喜欢这个现代化的C语言有两个原因。首先，对于学习一种大的语言，如C++，我太弱智。生活似乎只是充斥了需要理解的更有趣的东西。第二，我觉得手动控制的这一特定层次让我速度更快地产生更好的效果。

这里的要点不是C与C++，而是手动控制对高端专业用户的价值。这不是偶然，世界上最好的汽车、照相机和咖啡机都有手动控制。在现场微调这一层次，往往导致世界级的成功和不太成功之间的差异。 386

当你真的，真的关心序列化的速度和/或结果的大小（通常这些是相互矛盾的）时，你需要手写的二进制序列化。换句话说，让我们来听听"讨厌"先生怎么说！

编写一个高效讨厌的编码器/解码器（CODEC）的基本流程是：

- 构建有代表性的数据集和测试应用程序，它们可以对编解码器进行压力测试。
- 编写编解码器的第一个哑巴版本。
- 测试、测量、改进和重复，直到你用完了时间和/或金钱。

下面是一些用来改善我们的编解码器的技术：

- *使用剖析器*。原因很简单，没有办法知道你的代码在做什么，除非你已经剖析出它的函数执行次数和每个函数的CPU开销。当你发现代码的热点后，设法解决这些问题。
- *消除内存分配*。在现代的Linux内核中，堆是非常快的，但它仍然是最简陋的编解码器的瓶颈。在较旧版本的内核中，堆可能会非常慢，需要在代码中尽可能使用局部变量（栈）来取代堆。
- *在不同的平台上用不同的编译器和编译器选项测试*。除了堆之外，还有许多其他的差异。你需要学习主要的差异，并允许使用它们。
- *使用状态来更好地压缩*。如果你担心编解码器的性能，那么你几乎肯定是将相同类型的数据发送了很多次。数据实例之间将有冗余。你可以检测这些冗余并用它来压缩（例如，用一个短的值表示"与上次一样"）。
- *了解你的数据*。最好的压缩技术（在紧凑性的CPU的成本方面）需要知道数据的相关信息。例如，用于压缩一个单词列表、视频和股票市场数据流的技术都不同。
- *准备好打破规则*。你真的需要用大端网络字节顺序对整数编码吗？x86和ARM占

了几乎所有现代的 CPU 的数量，但它们使用小端字节顺序（ARM 实际上是双端的，但 Android 与 Windows 和 iOS 一样，是小端的）。

代码生成

看完前面的两节，你可能会很惊讶，“我可以自己编写出比通用的 IDL 生成器更好的东
387 西吗？”如果这种想法闯进你的心灵，它可能在不一会儿之后消失，被实际会涉及多少工作量的沮丧计算结果驱散。

如果我告诉你一个便宜且快速地创建自定义 IDL 生成器的方法会怎样？你可以有一种方式来获得完美的合同文档，符合你的需要和特定于领域的代码，而所有你需要做的是在这里签字放弃你的灵魂（*谁曾经真正使用它呢，我说得对吗？*）……

直到几年前，我们还在 iMatix 使用代码生成来构建比以往任何时候都更大、更雄心勃勃的系统，然后我们认为这种技术（生成器脚本语言，或 GSL）对于共同使用真是太危险了，因此我们封印存档并用沉重的锁链把它关在一个很深的地下城中。在现实中，我们实际上把它贴在 GitHub 上。如果你想尝试即将到来的例子，请抓取资源库（*https://github.com/imatix/gsl*），并构建自己的 *gsl* 命令。在 *src* 子目录中输入“make”应该可以做到这一点（如果你是热爱 Windows 的家伙，我敢肯定你会发送带项目文件的补丁）。

这一节根本不是针对 GSL 而言的，而是关于一个有用的和鲜为人知的技巧，它对希望提高自己以及他们的工作的雄心勃勃的架构师是很便利的。一旦你学会了窍门，就可以在很短的时间内自己造出代码生成器。大多数软件工程师了解的代码生成器来自一个单一的硬编码模式。例如，Ragel“从正规语言编译可执行的有限状态机”（即 Ragel 的模型是一个正规语言）。这当然适用于一个好的问题集，但它远未普及。你如何用 Ragel 来描述一个 API 呢？或一个项目的 makefile？甚至是一个类似我们在第 4 章用来设计双星模式的有限状态机呢？

所有这些都将受益于代码生成，但没有普遍的模式。因此，关键是要根据你的需要设计自己的模型，然后再把编码生成器作为针对这些模型的廉价的编译器。你需要有一些做好模型的经验，也需要一种能够廉价地构建定制的代码生成器的技术。诸如 Perl 和 Python 脚本语言都是不错的选择。然而，我们实际上专门针对这个构建了 GSL，而这就是我喜欢的方式。

让我们举一个简单的例子，它关系到我们已经知道的知识。以后，我们将看到更广泛的例子，因为我真的相信，代码生成对于大规模的工作是至关重要的知识。在第 4 章中，我们制定了管家协议（MDP），并编写了客户端、代理和工人。现在，我们可以通过建立自己的接口描述语言和代码生成器来机械地生成这些部件吗？

当编写一个 GSL 模型时，我们可以使用任何我们喜欢的语义。换句话说，我们可以当场发明特定于领域的语言。我会发明两种语言，看你是否能猜出它们各代表什么：

388

```
slideshow
    name = Cookery level 3
    page
        title = French Cuisine
        item = Overview
        item = The historical cuisine
        item = The nouvelle cuisine
        item = Why the French live longer
    page
        title = Overview
        item = Soups and salads
        item = Le plat principal
        item = Béchamel and other sauces
        item = Pastries, cakes, and quiches
        item = Soufflé - cheese to strawberry
```

那么，下面这个如何？

```
table
    name = person
    column
        name = firstname
        type = string
    column
        name = lastname
        type = string
    column
        name = rating
        type = integer
```

我们可以把第一段代码编译成一个演示文稿。第二段代码可以编译成 SQL 来创建和处理一个数据库表。因此，对于这个练习，我们的领域语言——模型——是由“类”组成的，这些类包含由各种类型的“字段”组成的“消息”。我把这个例子故意设计成人们熟悉的样子。下面是 MDP 客户端协议：

```
<class name = "mdp_client">
    MDP/Client
    <header>
        <field name = "empty" type = "string" value = ""
            >Empty frame</field>
        <field name = "protocol" type = "string" value = "MDPC01"
            >Protocol identifier</field>
```

```
    </header>
    <message name = "request">
        Client request to broker
        <field name = "service" type = "string">Service name</field>
        <field name = "body" type = "frame">Request body</field>
    </message>
    <message name = "reply">
        Response back to client
        <field name = "service" type = "string">Service name</field>
        <field name = "body" type = "frame">Response body</field>
    </message>
</class>
```

389 而下面是 MDP 工人协议：

```
<class name = "mdp_worker">
    MDP/Worker
    <header>
        <field name = "empty" type = "string" value = ""
            >Empty frame</field>
        <field name = "protocol" type = "string" value = "MDPW01"
            >Protocol identifier</field>
        <field name = "id" type = "octet">Message identifier</field>
    </header>
    <message name = "ready" id = "1">
        Worker tells broker it is ready
        <field name = "service" type = "string">Service name</field>
    </message>
    <message name = "request" id = "2">
        Client request to broker
        <field name = "client" type = "frame">Client address</field>
        <field name = "body" type = "frame">Request body</field>
    </message>
    <message name = "reply" id = "3">
        Worker returns reply to broker
        <field name = "client" type = "frame">Client address</field>
        <field name = "body" type = "frame">Request body</field>
    </message>
    <message name = "hearbeat" id = "4">
        Either peer tells the other it’s still alive
    </message>
    <message name = "disconnect" id = "5">
        Either peer tells other the party is over
    </message>
</class>
```

GSL 使用 XML 作为它的建模语言。XML 有一个不佳的声誉，它已被太多企业的污水渠拖垮，变得臭名昭著，但只要你保持它的简单性，它还是有一些强大的优点的。任何用来编写条目和属性的自我描述的层次结构的方法都是可行的。

现在，这里显示的是一个简短的用 GSL 编写的 IDL 生成器，它把我们的协议模型转换成文档：

```
.#  简单的 IDL 生成器 (specs.gsl)
.#
.output "$(class.name).md"
## $(string.trim (class.?''):left) 协议 The $(string.trim (class.?''):left)
Protocol
.for message
.   frames = count (class->header.field) + count (field)

A $(message.NAME) command consists of a multipart message of $(frames)
frames:

.   for class->header.field
.       if name = "id"
* Frame $(item ()): 0x$(message.id:%02x) (1 byte, $(message.NAME))
.       else
* Frame $(item ()): "$(value:)" ($(string.length ("$(value)")) \
bytes, $(field.:))
.       endif
.   endfor
.   index = count (class->header.field) + 1
.   for field
* Frame $(index): $(field.?'') \
.       if type = "string"
(printable string)
.       elsif type = "frame"
(opaque binary)
.           index += 1
.       else
.           echo "E: unknown field type: $(type)"
.       endif
.       index += 1
.   endfor
.endfor
```
390

XML 模式和脚本都在 *examples/models* 子目录中。要执行代码生成，输入下面这个命令：

```
gsl -script:specs mdp_client.xml mdp_worker.xml
```

以下是我们得到的工人协议的 Markdown 文本：

```
## MDP/Worker 协议

READY（就绪）命令由 4 帧的多部分消息组成：

* 第 1 帧："" (0 字节，空帧)
* 第 2 帧："MDPW01" (6 个字节，协议标识符)
* 第 3 帧：0X01 (1 个字节，READY)
* 第 4 帧：服务名称（可打印字符串）

REQUEST（请求）命令由 5 帧的多部分消息组成：

* 第 1 帧："" (0 字节，空帧)
* 第 2 帧："MDPW01" (6 个字节，协议标识符)
* 第 3 帧：0X02 (1 个字节，REQUEST)
* 第 4 帧：客户端地址（不透明的二进制数据）
* 第 5 帧：请求正文（不透明的二进制数据）

REPLY（应答）命令由 5 帧的多部分消息组成：

* 第 1 帧："" (0 字节，空帧)
* 第 2 帧："MDPW01" (6 个字节，协议标识符)
* 第 3 帧：0X03 (1 个字节，REPLY)
* 第 4 帧：客户端地址（不透明的二进制数据）
* 第 5 帧：请求正文（不透明的二进制数据）

HEARBEAT（信号检测）命令由 3 帧的多部分消息组成：

* 第 1 帧："" (0 字节，空帧)
* 第 2 帧："MDPW01" (6 个字节，协议标识符)
* 第 3 帧：0x04 (1 个字节，HEARBEAT)

DISCONNECT（断开）命令由 3 帧的多部分消息组成：

* 第 1 帧："" (0 字节，空帧)
* 第 2 帧："MDPW01" (6 个字节，协议标识符)
* 第 3 帧：0X05 (1 个字节，DISCONNECT)
```

391

正如你可以看到的，这些文本和我原来的手写规范很接近。现在，如果你已经克隆了本书的存储库并且你正在查看 *examples/models* 中的代码，就可以生成 MDP 客户端和工人的编解码器。我们将相同的两个模型传递到不同的代码生成器：

```
gsl -script:codec_c mdp_client.xml mdp_worker.xml
```

这个命令产生了 `mdp_client` 和 `mdp_worker` 类。其实，MDP 是如此简单，以至于它几乎不值得付出编写代码生成器的努力。当我们想要改变协议（我们为独立的管家项目做的）时，好处就体现出来了。我们修改协议，并运行该命令，最终生成出更完美的代码。

codec_c.gsl 代码生成器并不简短，但产生的编解码器，比我原本为管家项目合在一起的手写代码要好得多。例如，手写代码没有任何错误检查，如果你给它传递假消息，那么它会崩溃。

我现在要解释 GSL 推动的面向模型的代码生成器的利弊。强大不是免费得来的，在我们的业务中最大的陷阱之一就是凭空创造概念的能力。GSL 使得这特别容易，因此它可能成为一个同样危险的工具。

不要发明概念。设计师的工作是消除问题，而不是增加新的功能。

首先，我将列出面向模型的代码生成器的优点：

- 你可以创建映射到你的现实世界的“完美”抽象。所以，我们的协议模型百分之百地映射管家的“现实世界”。若没有以任何方式调整和改变模型的自由，这将是不可能的。
- 你可以快速、廉价地开发这些完美的模型。
- 你可以生成任何文本输出。可以从一个单一的模型创建文档并用任何语言编写代码、测试工具——最终你能想到的任何输出。
- 你可以生成（我的意思是严格的）*完美*输出，因为它可以廉价地将你的代码生成器改善到你想要的任何水平。
- 你获得了结合规范和语义的单一来源。 392
- 你可以利用它把一个小团队发展到巨大规模。在 iMatix，我们从包括代码生成脚本本身大约 85K 行的输入模型生成了百万行的 OpenAMQ 消息传递产品。

现在，让我们来看看它的缺点：

- 你对项目添加了对工具的依赖。
- 你可能会忘乎所以，并纯粹由于生成它们的喜悦而建立模型。
- 你可能会疏远新人，而他们会从你的工作看到“奇怪的东西”。
- 你可能会为不对你的项目投资的人们找了一个强有力的借口。

玩世不恭地说，面向模型的滥用在如下环境中效果很不错：你需要产生大量的完美的代码，这些代码可以用很少的工作去维护，而且这些工作是*非你莫属*的。就个人而言，我喜欢跨过我的河流，继续前进。但如果你想要有长期工作保障，那么这个环境几乎是完美的。

所以，如果你使用 GSL，并希望围绕你的工作创建一个开放社区，下面是我的建议：

- 只在你如果不这么做就将手工编写烦人的代码时使用它。
- 自然的设计模式，是人们所期望看到的。
- 首先手工编写代码，以让你知道该生成什么。
- 不要过度使用它。保持简单！不要太形而上学！
- 将它逐渐引入到项目中。
- 将生成的代码放到你的存储库中。

我们已经在一些围绕 ØMQ 的项目中使用了 GSL。例如，高层次 C 绑定程序，CZMQ，使用 GSL 来生成套接字选项类（zsockopt）。一个 300 行的代码生成器将 78 行的 XML 模型变成 1500 行的完美但很无聊的代码。这是一种完胜。

传输文件

让我们暂停讲课，休息一下，并回到我们的最爱和做所有这些事情的初衷：代码。

"我如何发送文件？"是 ØMQ 邮件列表中一个常见的问题。毫不奇怪，文件传输也许是
393 最古老、最明显的一类消息传递。ØMQ 在发送事件和任务方面是预制得很不错的，但它不擅长发送文件。

我已经答应了，在一年或两年内，写出一个适当的解释。这里有一个无偿的资料片，它能照亮你的早晨：单词 *proper* 来自古老的法语 *propre*，它意味着"干净"。不熟悉使用热水和肥皂的黑暗时代的英国庶民将这个词的意思修改为"外国"或"上层阶级"，如"这是上层阶级的食物！"后来它的意思变成只是"真实"，如"你把我们放进的真是一个烂摊子！"

因此，让我们研究文件传输。为何你不能只是拿起一个随机文件，蒙上它的眼睛，将它整个推入一个消息呢？这有几个原因。最明显的原因是，尽管几十年来在 RAM 的容量上有确定的增长（而我们中的老前辈谁不怀念存钱买 1024 字节的内存扩展卡的旧时光呢），磁盘容量始终保持比内存大得多。即使我们可以用一条指令（例如，使用一个类似 *sendfile* 的系统调用）发送一个文件，也会面临网络既不是无限快的，也不完全可靠的现实。尝试过在 WiFi 或任何一种缓慢的片状网络上几次上传一个大文件后，你就会认识到一个适当的文件传输协议需要有一种从故障中恢复的方法。即，它需要有一个方法来只发送文件的尚未接收到的部分。

最后，经过这一切，如果你建立了一个适当的文件服务器，你就会发现，简单地将大量数据发送到许多客户端会产生下面这种情况，在技术的说法下，我们喜欢把它叫作"由

于所有可用堆内存被一个设计得糟糕的应用程序耗尽而导致的服务器崩溃”。一个适当的文件传输协议，需要注意内存的使用。

我们将正确地逐步解决这些问题，这应该有希望产生一个在 ØMQ 上运行良好和正确的文件传输协议。首先，让我们用随机数据产生 1GB 的测试文件（译者注：指这个 1GB=2 的 30 次方字节，而不是 10 的 9 次方字节，但实际上，业界内存容量一般就是按 2 的幂算的，硬盘容量才是按 10 的幂算的）：

```
dd if=/dev/urandom of=testdata bs=1M count=1024
```

当我们有很多客户端同时请求同一个文件的时候，这个文件大得足以造成麻烦，并在许多机器上，无论如何，1GB 的内存都会因为太大了而无法分配成功。作为基本参考，让我们衡量将此文件从磁盘复制到磁盘需要多长时间。这将告诉我们，我们的文件传输协议额外增加了多少（包括网络开销）：

```
$ time cp testdata testdata2

real    0m7.143s
user    0m0.012s
sys     0m1.188s
```

四位数精度的时间是一种误导，预计有上下 25% 的波动。这只是在“数量级”上准确的测量结果。

示例 7-1 显示了我们的代码初稿，其中客户端请求测试数据，而服务器只是把它当作一系列的消息一气呵成地发送，其中每个消息各执一个“块”。 394

示例7-1：文件传输测试，模型1（fileio1.c）

```
//  文件传输模型 #1
//
//  其中，服务器用大块把整个文件发送到客户端，
//  没有采取流量控制。

#include <czmq.h>
#define CHUNK_SIZE  250000

static void
client_thread (void *args, zctx_t *ctx, void *pipe)
{
    void *dealer = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (dealer, "tcp://127.0.0.1:6000");

    zstr_send (dealer, "fetch");
```

```
    size_t total = 0;           //  接收的总字节数
    size_t chunks = 0;          //  接收的总块数

    while (true) {
        zframe_t *frame = zframe_recv (dealer);
        if (!frame)
            break;              //  关闭，退出
        chunks++;
        size_t size = zframe_size (frame);
        zframe_destroy (&frame);
        total += size;
        if (size == 0)
            break;              //  接收了整个文件
    }
    printf ("%zd chunks received, %zd bytes\n", chunks, total);
    zstr_send (pipe, "OK");
}

static void
free_chunk (void *data, void *arg)
{
    free (data);
}
```

如示例 7-2 所示，服务器线程从磁盘中读取文件块，并将每个块作为一个单独的消息发送给客户端。因为我们只有一个测试文件，所以我们会打开它一次，然后根据需要处理它。

395 示例7-2：文件传输测试，模型1（fileio1.c）：文件服务器线程

```
static void
server_thread (void *args, zctx_t *ctx, void *pipe)
{
    FILE *file = fopen ("testdata", "r");
    assert (file);

    void *router = zsocket_new (ctx, ZMQ_ROUTER);
    //  HWM 的默认值是 1000，这将丢弃在这里的消息，
    //  因为我们发送 1000 多块的测试数据，
    //  所以设置一个无限的 HWM 是一种简单而愚蠢的解决方案
    zsocket_set_hwm (router, 0);
    zsocket_bind (router, "tcp://*:6000");
    while (true) {
        //  每个消息的第 1 帧是发送者的身份
        zframe_t *identity = zframe_recv (router);
        if (!identity)
            break;              //  关闭，退出
```

```
        // 第2帧是“fetch”命令
        char *command = zstr_recv (router);
        assert (streq (command, "fetch"));
        free (command);

        while (true) {
            byte *data = malloc (CHUNK_SIZE);
            assert (data);
            size_t size = fread (data, 1, CHUNK_SIZE, file);
            zframe_t *chunk = zframe_new_zero_copy (data, size, free_chunk,NULL);
            zframe_send (&identity, router, ZFRAME_REUSE + ZFRAME_MORE);
            zframe_send (&chunk, router, 0);
            if (size == 0)
                break;          // 始终结束于一个大小为零的帧
        }
        zframe_destroy (&identity);
    }
    fclose (file);
}
```

其主任务，如示例 7-3 所示，启动客户端和服务器的线程，比起用多个进程来测试，用带有多个线程的单个进程来测试更容易些。

示例7-3：文件传输测试，模型1（fileio1.c）：文件主线程

```
int main (void)
{
    // 启动子线程
    zctx_t *ctx = zctx_new ();
    zthread_fork (ctx, server_thread, NULL);
    void *client =
    zthread_fork (ctx, client_thread, NULL);
    // 循环，直到客户端告诉我们它已完成
    char *string = zstr_recv (client);
    free (string);
    // 清除服务器线程
    zctx_destroy (&ctx);
    return 0;
}
```

396

这很简单，但我们已经遇到了一个问题：如果我们发送过多的数据到 ROUTER 套接字，很容易就会溢出。简单而愚蠢的解决办法是把一个无限的高水位标记放在套接字中。这是愚蠢的，因为我们现在已经失去防止耗尽服务器内存的保护措施。然而，如果没有一个无限的 HWM，我们又可能会丢失大量的文件块。

试试这个：把 HWM 设置为 1000（在 ØMQ v3.x 版本，这是默认的），然后将块大小减小为 100K，所以我们一次传送 10K 个块。运行测试，你会看到它永远都不会执行完成。因为 zmq_socket() 手册页写道，对于 ROUTER 套接字："ZMQ_HWM 选项的操作：删除"。

我们必须预先控制服务器发送的数据量。发送超过网络处理能力的数据量是没有意义的。让我们尝试一次发送一个块。在这个版本的协议中，客户端会明确地说，"给我块 N"，而服务器会从磁盘读取特定块并将其发送。

示例 7-4 给出了改进的第二个模型，其中客户端一次请求一个块，而服务器只发送从客户端获得的每一个请求索要的一个块。

示例7-4：文件传输测试，模型2（fileio2.c）

```
//  文件传输测试，模型 #2
//
//  其中客户端分别请求每个块，这样
//  就消除了服务器队列溢出，但速度有所下降。

#include <czmq.h>
#define CHUNK_SIZE  250000

static void
client_thread (void *args, zctx_t *ctx, void *pipe)
{
    void *dealer = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_set_hwm (dealer, 1);
    zsocket_connect (dealer, "tcp://127.0.0.1:6000");

    size_t total = 0;           //  接收的总字节数
    size_t chunks = 0;          //  接收的总块数

    while (true) {
        //  请求下一个块
        zstr_sendfm (dealer, "fetch");
        zstr_sendfm (dealer, "%ld", total);
        zstr_sendf  (dealer, "%ld", CHUNK_SIZE);

397     zframe_t *chunk = zframe_recv (dealer);
        if (!chunk)
            break;              //  关闭，退出
        chunks++;
        size_t size = zframe_size (chunk);
        zframe_destroy (&chunk);
        total += size;
        if (size < CHUNK_SIZE)
```

```
            break;              //  接收了最后一个块，退出
    }
    printf ("%zd chunks received, %zd bytes\n", chunks, total);
    zstr_send (pipe, "OK");
}

static void
free_chunk (void *data, void *arg)
{
    free (data);
}
```

服务器线程（参见示例 7-5）等待来自客户端的块请求，读取数据块，并将其发送回客户端。

示例7-5：文件传输测试，模型2（fileio2.c）：文件服务器线程

```
static void
server_thread (void *args, zctx_t *ctx, void *pipe)
{
    FILE *file = fopen ("testdata", "r");
    assert (file);

    void *router = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_hwm (router, 1);
    zsocket_bind (router, "tcp://*:6000");
    while (true) {
        //  每个消息的第 1 帧是发送者身份
        zframe_t *identity = zframe_recv (router);
        if (!identity)
            break;              //  关闭，退出

        //  第 2 帧是“fetch”命令
        char *command = zstr_recv (router);
        assert (streq (command, "fetch"));
        free (command);

        //  第 3 帧是块在文件中的偏移量
        char *offset_str = zstr_recv (router);
        size_t offset = atoi (offset_str);
        free (offset_str);

        //  第 4 帧是块的最大尺寸
        char *chunksz_str = zstr_recv (router);
        size_t chunksz = atoi (chunksz_str);
        free (chunksz_str);

        // 从文件读取数据块
```
398

```
        fseek (file, offset, SEEK_SET);
        byte *data = malloc (chunksz);
        assert (data);

        //  将得到的块发送到客户端
        size_t size = fread (data, 1, chunksz, file);
        zframe_t *chunk = zframe_new_zero_copy (data, size, free_chunk, NULL);
        zframe_send (&identity, router, ZFRAME_MORE);
        zframe_send (&chunk, router, 0);
    }
    fclose (file);
}

//  主任务与第一个模型中的完全相同
...
```

现在速度慢得多了，这是因为客户端和服务器之间的往复交流。在本地环路连接（客户端和服务器在同一台电脑）上，每个请求 - 应答来回大约耗费 300 微秒。这听起来并不是很多，但它会很快就会累加起来：

```
$ time ./fileio1
4296 chunks received, 1073741824 bytes

real    0m0.669s
user    0m0.056s
sys     0m1.048s

$ time ./fileio2
4295 chunks received, 1073741824 bytes

real    0m2.389s
user    0m0.312s
sys     0m2.136s
```

这里有两个宝贵的教训。首先，虽然请求 - 应答是容易的，但对于大容量数据流，它的速度也太慢了。虽然一次性耗费 300 微秒是可以的。但为每个块都耗费这么多时间是不能接受的，尤其是在延迟大约高 1000 倍的实际网络中。

第二点我之前已经说过了，但还是要再次重复：对 ØMQ 之上的一个协议进行实验、测量和改进简单得令人难以置信。而当某个东西的成本一路下跌时，你可以供得起的会多很多。不要学着去孤立地开发和证明你的协议：我见过有的团队浪费了很多时间试图改
399 善设计不当的协议，但这些协议过于深入地嵌入到应用程序，从而难以测试或修复。

除了性能以外，我们的模型 2 文件传输协议并没有那么糟糕，这体现在如下方面：

- 它完全消除了内存耗尽的风险。为了证明这一点，我们在发送方和接收方都把高水位标记设置为 1。
- 它允许客户端选择块大小，这是非常有用的，因为如果有对块大小进行的调优（为网络条件，为文件类型，或者为了进一步减少内存消耗），那么应该由客户端做这件事。
- 它给我们提供了能完全重新启动的文件传输。
- 它允许客户端在任何时间点取消文件传输。

如果有个协议允许我们不必对每个块都执行一次请求，那么它会非常有用。我们需要的是这样一种方法，它让服务器发送多个数据块，而不必等待客户端请求或者确认每一个。我们的选择是什么呢？

- 服务器可以一次发送 10 个块，然后等待一个确认。但这正如将块大小乘以 10，所以它是没有意义的。是的，它正如所有 10 的倍数那样毫无意义。
- 服务器可以发送数据块而无须客户端的任何回应，但在每次发送之间有一个轻微的延迟，所以它发送块的速度最快也不会超出网络的处理能力。但是，这就要求服务器知道在网络层发生了什么，这听起来像艰巨的工作。这也可怕地破坏了分层。并且如果网络非常快，但客户端本身却很慢，会发生什么情况？又该在哪里对块进行排队呢？
- 服务器可以尝试窥探发送队列，即，看它有多满，并仅当队列没有满时才发送。但是 ØMQ 不允许这么做，因为这是行不通的，与节流不起作用是一样的道理。服务器和网络可能会比足够快还快，但客户端可能是一个缓慢的小设备。
- 我们可以修改 `libzmq`，使它对到达 HWM 采取其他一些行动。也许它可以阻塞？这将意味着一个单独的缓慢客户端将阻塞整个服务器，所以不用了，谢谢。也许它可以给调用者返回一个错误？然后服务器可以做一些聪明的像 …… 嗯，确实它可以做的任何东西都不会比删除消息更好。

除了复杂，各种不愉快，这些选项甚至都不能工作。我们需要的是一种让客户端在后台异步地告诉服务器它已准备就绪的方法。我们需要某种异步的流量控制。如果我们正确地这样做了，数据应该会没有中断地从服务器流到客户端，但仅在客户端阅读它时。让我们回顾一下我们的协议。这是第一个协议：

400

```
C: fetch
S: chunk 1
S: chunk 2
S: chunk 3
...
```

而第二个协议引入了对每个块的请求：

```
C: fetch chunk 1
S: send chunk 1
C: fetch chunk 2
S: send chunk 2
C: fetch chunk 3
S: send chunk 3
C: fetch chunk 4
...
```

现在，神秘地挥挥手，这里有一个修改后的协议，它解决了性能问题：

```
C: fetch chunk 1
C: fetch chunk 2
C: fetch chunk 3
S: send chunk 1
C: fetch chunk 4
S: send chunk 2
S: send chunk 3
....
```

它看起来可疑地类似于第二个协议。事实上，除了我们发送多个请求，而无须等待每一个请求的应答，它们基本上是相同的。这是一种叫作“流水线”的技术，并且因为我们的 DEALER 和 ROUTER 套接字是完全异步的，所以它能够工作。

示例 7-6 给出了文件传输测试平台的第三种模型，它采用流水线。客户端提前发送多个请求（“信用”），然后每次处理一个输入块时，它都会再发送一个或多个信用。服务器不会发送比客户端所要求的更多的块。

示例7-6：文件传输测试，模型3（fileio3.c）

```
//  文件传输模型 #3
//
//  其中客户端分别请求每个块，使用
//  命令流水线给我们一个基于信用的流量控制。

#include <czmq.h>
#define CHUNK_SIZE  250000
#define PIPELINE    10

static void
client_thread (void *args, zctx_t *ctx, void *pipe)
{
401     void *dealer = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (dealer, "tcp://127.0.0.1:6000");
```

```
    // 至此，已经有很多块在传输中
    size_t credit = PIPELINE;

    size_t total = 0;           // 接收的总字节数
    size_t chunks = 0;          // 接收的总块数
    size_t offset = 0;          // 下一个请求块的偏移量

    while (true) {
        while (credit) {
            // 请求下一个块
            zstr_sendfm (dealer, "fetch");
            zstr_sendfm (dealer, "%ld", offset);
            zstr_sendf  (dealer, "%ld", CHUNK_SIZE);
            offset += CHUNK_SIZE;
            credit--;
        }
        zframe_t *chunk = zframe_recv (dealer);
        if (!chunk)
            break;              // 关闭，退出
        chunks++;
        credit++;
        size_t size = zframe_size (chunk);
        zframe_destroy (&chunk);
        total += size;
        if (size < CHUNK_SIZE)
            break;              // 接收了最后一个块，退出
    }
    printf ("%zd chunks received, %zd bytes\n", chunks, total);
    zstr_send (pipe, "OK");
}

// 代码的其余部分与模型 2 是完全一样的，除了
// 我们在到 PIPELINE 的服务器的 ROUTER 接口上设定 HWM
// 作为一种健全检查
...
```

这个技巧让我们能完全控制端到端的管道，包括所有的网络缓冲区和在发送者与接收者上的 ØMQ 队列。我们保证管道始终充满了数据，但数据从来不会增长到超出预定义的限制。更重要的是，由客户端决定什么时候发送一个“信用”给发送者。这个时候可以是当它接收到一个块时，或当它已经完全处理一个块时。这是异步发生的，没有显著的性能开销。

在第三个模型中，我选择了大小为 10 条消息的管道（每个消息都是一个块）。这将花费每个客户端最多 2.5MB 的内存，所以用 1GB 的内存，我们至少可以处理 400 个客户端，可以尝试来计算理想的管道大小。发送 1GB 的文件大约需要 0.7 秒，这表明处理一个块
402 大约需要 160 微秒。一个来回需要 300 微秒，所以管道至少需要 3 ～ 5 个消息来保持服务器繁忙。在实践中，使用大小为 5 条消息的管道，我仍然得到性能峰值，可能是因为信用的消息有时会被传出的数据延迟。在大小为 10 条消息时，这个程序始终如一地工作：

```
$ time ./fileio3
4291 chunks received, 1072741824 bytes

real    0m0.777s
user    0m0.096s
sys     0m1.120s
```

请严格地执行测量。你的计算可能是好的，但现实世界往往有自己的看法。

我们所做的显然还不是一个真正的文件传输协议，但它证明了模式，我认为这是最简单可行的设计。对于一个真正的能工作的协议，我们可能要添加以下内容中的一些或全部：

- 身份验证和访问控制，即使没有加密：重点不是保护敏感数据，而是要捕获类似于将测试数据发送到生产服务器的错误。
- 一种廉价的请求，包括文件路径，可选的压缩，和我们从 HTTP 学到的确实有用的其他东西（如 `If-Modified-Since`）。
- 一种廉价的响应，至少对于第一个块，提供了元数据，如文件大小（这样客户端就可以预先配置，避免不愉快的磁盘已满的情况）。
- 一气呵成地获取一组文件的能力，否则该协议对于大套的小文件就将变得效率低下。
- 当客户端完全接收一个文件时，它发出确认，这样，如果客户端意外断开，就能对可能丢失的块执行恢复。

到目前为止，我们的语义一直是“获取”，也就是说，接受者（由于某种原因）知道，它需要一个特定的文件，因此它要求得到它。其中哪个文件存在，以及它们在哪里的知识，随后在带外被传递出去（例如，在 HTTP 中，通过在 HTML 页面中的链接）。

“推送”的语义会怎么样呢？这有两种可能的用例。第一种，如果我们采用集中式架构，其中文件在一台主要的“服务器”上（这不是我提倡的，但人们有时会这样做），那么这对于允许客户端上传文件到服务器是非常有用的。第二种，它让我们对文件做一种发布 - 订阅的操作，其中客户端请求某一个类型的所有新文件，当服务器得到这些时，它将其转发给客户端。

获取语义是同步的，而推送语义是异步的，异步速度更快。此外，你可以制作类似“订

阅这个路径”的可爱的东西，从而创建发布 - 订阅文件传输架构。这是如此明显地优秀，所以我应该不需要解释它解决什么问题。

尽管如此，下面是获取语义存在的问题：要用带外途径来告诉客户端什么文件存在。不 403
管你如何做到这一点，它最终都会很复杂。无论是客户端必须轮询，还是你需要一个单独的发布 - 订阅通道来保持客户端最新，还是你需要用户交互，都不例外。

但是，经过稍微多一点的思考，我们可以看到，获取只是发布 - 订阅的一个特例。因此，我们可以得到两全其美的结果。下面是一般的设计：

- 取此路径
- 这里是信用（重复）

为了使这能够工作（我们将会做到的，我亲爱的读者），我们需要对于如何把信用发送到服务器更明确一点。将管道化的“获取块”的请求作为信用处理的可爱技巧不会生效，因为客户端不再知道什么文件确实存在，它们有多大，或任何的东西。如果客户端说：“我擅长处理 250,000 字节的数据”，这应该对 250K 字节的 1 个文件，或 2500 字节的 100 个文件同样有效。

这给了我们“基于信用的流量控制”，从而有效地消除了对高水位标记的需求和内存溢出的风险。

状态机

软件工程师倾向于把（有限）状态机当作一种中介解释器。也就是说，你把一个正规的语言编译成一个状态机，然后执行该状态机。开发者很少看到状态机本身：这是一个内部表示——优化、压缩和离奇的。

然而，事实证明，状态机作为一流的建模语言，它对协议处理程序，如 ØMQ 客户端和服务器也是有价值的。ØMQ 使得协议比较容易设计，但我们从来没有定义一种良好的模式来恰当地编写这些客户端和服务器。

一种协议，至少包含两个层次：

- 如何表示在线路上的单个消息。
- 消息如何在节点之间流动，以及每个消息的意义。

我们已经在这一章看到了如何产生负责处理序列化的编解码器，这是一个良好的开端。但是，如果我们把第二项工作留给开发者，这就会给他们带来很大的解释余地。当我们制作更雄心勃勃的协议（文件传输 + 信号检测 + 信用 + 认证）时，试图手工来实现客户

端和服务器就变得越来越不理智了。

404 是的，人们几乎是机械地做到这一点的。但这么做的成本很高，而本来是可以避免的。在本节中，我将解释如何使用状态机对协议建模，以及如何从这些模型产生整齐而可靠的代码。

我使用状态机作为软件构造工具的初步经验可以追溯到 1985 年，我第一份真正为应用程序开发人员制作工具的工作。1991 年，我把这些知识变成一个称为 Libero 的免费软件工具，它从一个简单的文本模型吐出可执行的状态机。

关于 Libero 的模型的重点是，它是可读的。也就是说，你把你的程序逻辑描述成命名的状态，每个状态接受一组事件，各做一些实际的工作。由此产生的状态机挂接到应用程序代码中，并像一个老板一样驱动它。

Libero 迷人地擅长它的工作，精通多种语言，并谦虚地流行，给出状态机的神秘性质。我们在几十种大型的分布式应用程序中狂热地使用 Libero，其中一个经过 20 年的运作后终于在 2011 年被关掉。状态机驱动的代码构建工作得非常好，这是令人印象深刻的，因为这种做法从未进入软件工程的主流。

因此，在本节，我要解释 Libero 的模型，并展示如何使用它来生成 ØMQ 客户端和服务器。我们将再次使用 GSL，但就像我说的，原则是通用的，而你可以使用任何脚本语言和代码生成器联用。

作为一个可以工作的例子，让我们看看如何使用 ROUTER 套接字上的节点承载有状态的对话。我们将使用状态机开发一个服务器（和手工开发一个客户端）。我们有一个称作“NOM”的简单协议，我会使用非常严肃的“Unprotocol 的关键字”（*http://unprotocols.org/blog:2*）建议：

```
nom-protocol    = open-peering *use-peering

open-peering    = C:OHAI ( S:OHAI-OK / S:WTF )

use-peering     = C:ICANHAZ
                / S:CHEEZBURGER
                / C:HUGZ S:HUGZ-OK
                / S:HUGZ C:HUGZ-OK
```

我还没有找到一个用来解释状态机编程本质的快速方法。根据我的经验，它总是需要练习几天。对这个想法经过三到四天接触后，当某个东西在大脑中将各个部分连接在一起时，几乎可听到“咔嗒！”一声。我们将通过查看 NOM 服务器的状态机使之具体化。

关于状态机的一件有用的事情是，你可以逐个读取它们的状态。每个状态都有一个唯一 405
的描述性名称和一个或多个事件，这是我们以任何顺序列出的。对于每个事件，我们执行零个或多个动作，然后转移到下一个状态（或保持在相同的状态）。

在 ØMQ 协议的服务器中，我们对每个客户端都有一个状态机实例。这听起来很复杂，但其实不复杂，正如我们即将看到的。我们描述了我们的第一个状态（开始）为具有一个有效的事件“OHAI”。我们检查用户的凭据，然后到了身份验证的状态（参见图 7-1）。

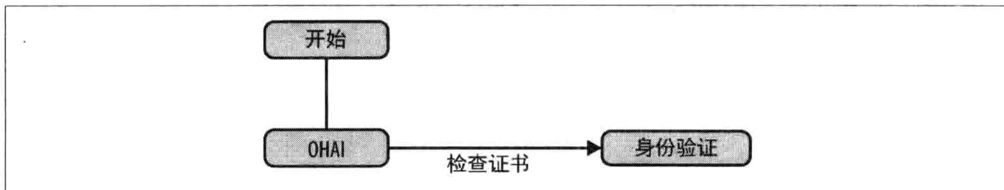


图7-1：“开始”状态

检查凭据动作会产生要么是“OK”要么是“出错”的事件。我们在身份验证的状态，通过发送一个适当的应答返回给客户端来处理这两个可能的事件（参见图 7-2）。如果身份验证失败，我们回到开始状态，客户端可以再次尝试。

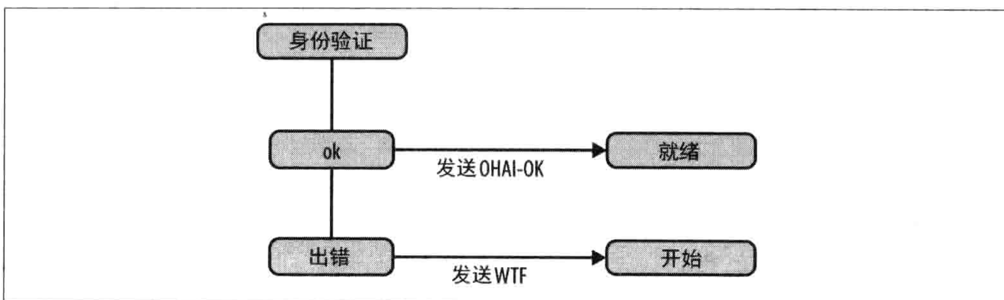


图7-2：“身份验证”状态

如果认证成功，我们就到达就绪状态。在这里，有三个可能的事件：来自客户端的 ICANHAZ 或 HUGZ 消息或信号检测（heartbeat）定时器事件（参见图 7-3）。

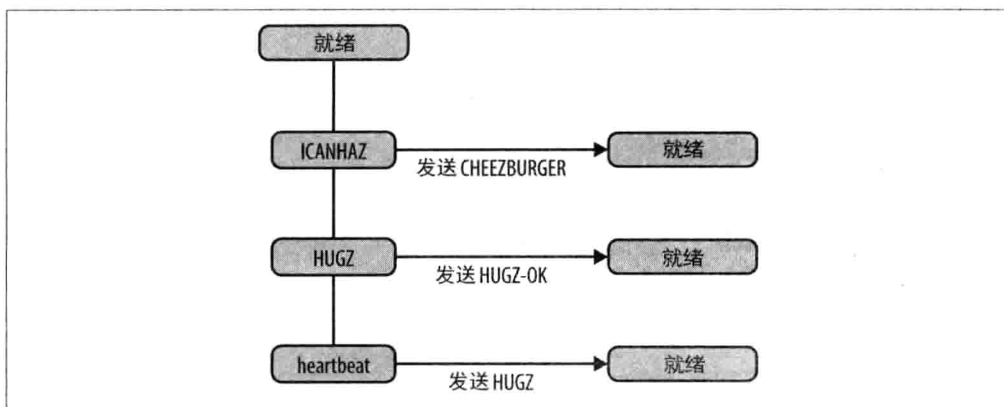


图7-3："就绪"状态

406 关于这个状态机模型，有几件事情是值得了解的：

- 大写的事件（如"HUGZ"）是作为消息来自客户端的"外部事件"。
- 小写事件（如"heartbeat"）是"内部事件"，由代码在服务器中产生。
- "发送 SOMETHING"动作是发送一个特定的应答返回给客户端的简写。
- 未在一个特定的状态中定义的事件被默默地忽略。

因此，上面这些漂亮的图片的原始来源就是一个 XML 模型：

```
<class name = "nom_server" script = "server_c">

<state name = "start">
    <event name = "OHAI" next = "authenticated">
        <action name = "check credentials" />
    </event>
</state>

<state name = "authenticated" >
    <event name = "ok" next = "ready" >
        <action name = "send" message = "OHAI-OK" />
    </event>
    <event name = "error" next = "start">
        <action name = "send" message = "WTF" />
    </event>
</state>

<state name = "ready">
    <event name = "ICANHAZ">
        <action name = "send" message = "CHEEZBURGER" />
```

407

```
    </event>
    <event name = "HUGZ">
        <action name = "send" message = "HUGZ-OK" />
    </event>
    <event name = "heartbeat">
        <action name = "send" message = "HUGZ" />
    </event>
</state>
</class>
```

代码生成器在 *examples/models/server_c.gsl* 中。这是我所使用并扩大为以后更严肃的工作的一个相当完整的工具。它将产生：

- 实现整个协议流的 C（*nom_server.c*、*nom_server.h*）服务器类。
- 一个运行在 XML 文件中列出的自检步骤的 selftest 方法。
- 图形形式的文档（漂亮的图片）。

下面是启动生成 NOM 服务器的一个简单的主程序：

```
#include "czmq.h"
#include "nom_server.h"

int main (int argc, char *argv [])
{
    printf ("Starting NOM protocol server on port 5670...\n");
    nom_server_t *server = nom_server_new ();
    nom_server_bind (server, "tcp://*:5670");
    nom_server_wait (server);
    nom_server_destroy (&server);
    return 0;
}
```

生成的 nom_server 类是相当经典的模型。它在 ROUTER 套接字上接受客户端的消息，因此每个请求的第一帧是客户端的连接身份。服务器管理一组客户端，每个客户端都有状态。当消息到达时，它将这些作为“事件”馈入状态机。下面是状态机的核心，它是 GSL 命令和我们打算产生的 C 代码的混合：

```
client_execute (client_t *self, int event)
{
    self->next_event = event;
    while (self->next_event) {
        self->event = self->next_event;
        self->next_event = 0;
        switch (self->state) {
```

```
.for class.state
            case $(name:c)_state:
.   for event
.       if index () > 1
                else
.       endif
                if (self->event == $(name:c)_event) {
.       for action
.           if name = "send"
                    zmsg_addstr (self->reply, "$(message:)");
.           else
                $(name:c)_action (self);
.           endif
.       endfor
.       if defined (event.next)
                    self->state = $(next:c)_state;
.       endif
                }
.   endfor
                break;
.endfor
        }
        if (zmsg_size (self->reply) > 1) {
            zmsg_send (&self->reply, self->router);
            self->reply = zmsg_new ();
            zmsg_add (self->reply, zframe_dup (self->address));
        }
    }
}
```

408

每个客户端被保存为带有各种属性的对象，包括我们需要用来表示一个状态机实例的变量：

```
event_t next_event;             //  下一个事件
state_t state;                  //  当前状态
event_t event;                  //  当前事件
```

现在你将看到，我们正在生成在技术上完美的代码，它具有精确的设计和我们希望的形状。唯一暗示了 nom_server 类不是手写代码的线索是，这段代码太好了。抱怨代码生成器生成的代码不好的人自己常常就是不好的代码生成器。

当我们需要扩展模型时，这项工作是微不足道的。例如，下面是我们产生自测代码的方法。首先，把一个“selftest”项添加到状态机，并编写我们的测试。我们不使用任何 XML 语法或验证，所以它确实只是一个打开编辑器并添加几行文本的问题：

```
<selftest>
    <step send = "OHAI" body = "Sleepy" recv = "WTF" />
    <step send = "OHAI" body = "Joe" recv = "OHAI-OK" />
    <step send = "ICANHAZ" recv = "CHEEZBURGER" />
    <step send = "HUGZ" recv = "HUGZ-OK" />
    <step recv = "HUGZ" />
</selftest>
```

凭空地设计，我决定，“send”和“recv”是表达“发送此请求，然后期望此应答”的一 409
种不错的方式。下面是将这个模型转化为实际代码的 GSL 代码：

```
.for class->selftest.step
.   if defined (send)
    msg = zmsg_new ();
    zmsg_addstr (msg, "$(send:)");
.       if defined (body)
    zmsg_addstr (msg, "$(body:)");
.       endif
    zmsg_send (&msg, dealer);

.   endif
.   if defined (recv)
    msg = zmsg_recv (dealer);
    assert (msg);
    command = zmsg_popstr (msg);
    assert (streq (command, "$(recv:)"));
    free (command);
    zmsg_destroy (&msg);

.   endif
.endfor
```

最后，任何状态机生成器更为棘手，但绝对重要的部分之一是我如何将其插入自己的代码？在本练习中，作为一个最小的例子，我想通过接受所有来自我的朋友 Joe（Joe 你好！）的 OHAI，并拒绝其他人的 OHAI 来实现“检查证书”的动作。经过一番思考，我决定直接从状态机模型，即 XML 文件中嵌入的动作正文抓取代码。所以，在 *nom_server.xml* 中，你会看到如下内容：

```
<action name = "check credentials">
    char *body = zmsg_popstr (self->request);
    if (body && streq (body, "Joe"))
        self->next_event = ok_event;
    else
        self->next_event = error_event;
    free (body);
```

```
</action>
```

而代码生成器抓取该 C 代码，并将其插入到生成的 *nom_server.c* 文件中：

```
.for class.action
static void
$(name:c)_action (client_t *self) {
$(string.trim (.):)
}
.endfor
```

410 现在，我们有一些非常优雅的东西：描述我的服务器状态机的单个源文件，并且还包含了我的动作的原生实现。高层次和低层次的一个不错的组合，它比 C 代码的规模小大约 90%。

当你的头脑定在你能用这样的杠杆产生出惊人的所有东西的概念时，请当心：虽然这种方法给你真正的力量，但它也驱使你远离你的同行，如果你走得太远，你就会发现自己的工作很孤独。

顺便说一句，这个简单的小状态机的设计正好对我们的自定义代码公开三个变量：

- `self->next_event`
- `self->request`
- `self->reply`

在 Libero 状态机模型中有几个我们并没有在这里使用的概念，但在我们编写更大的状态机的时候需要它们：

- 异常，这让我们写更简洁的状态机。当一个动作引发了异常，就停止对事件进一步处理。状态机随后可以定义如何处理异常事件。
- 默认状态，我们可以在这里定义默认的处理事件（对异常事件特别有用）。

使用 SASL 认证

在 2007 年设计 AMQP 的时候，我们选择的验证层是简单身份验证和安全层（*http://en.wikipedia.org/wiki/Simple_ Authentication_and_Security_Layer*）（SASL），这是我们从 BEEP 协议框架借鉴的一个想法。SASL 第一眼看起来很复杂，但它实际上是简单地、整齐地适合进入一个 ØMQ 为基础的协议中的。我特别喜欢 SASL 的一点是它的可扩展性。一开始可以使用匿名访问或纯文本身份验证并且没有安全性，随着时间的推移，可以在不改变你的协议的条件下发展到更安全的机制。

我现在不会给出一个深入的解释，因为稍后我们将在实践中看到 SASL。但我会解释其中的原则，以便你有所准备。

在 NOM 协议中，客户端以 OHAI 命令启动，服务器要么接受（“Joe 你好！”），要么拒绝该命令。这很简单，但不可扩展，因为服务器和客户端必须事先同意它们打算做什么样的身份验证。

SASL 引入的是一个天才的完全抽象与可谈判的安全层，它仍然很容易在协议层实现。它的工作原理如下：

1. 客户端连接。 411
2. 服务器质询客户端，传出它知道的一个安全“机制”列表。
3. 客户端从中选择它知道的一个安全机制，并用一个不透明的二进制大对象数据（而这里就是绝招）来回应服务器的质询，这个二进制大对象数据是一些通用的安全库计算并提供给客户端的。
4. 服务器采用客户端选择的安全性机制，以及数据的二进制大对象，并把它传递给其自身的安全库。
5. 该库要么接受客户端的回答，要么服务器再次质询。

存在许多免费的 SASL 库。当要进行实际的编码时，我们将只实现两个机制，ANONYMOUS 和 PLAIN，它们不需要任何特殊的库。

为了支持 SASL，我们必须在我们的“开放式对等”流中添加一个可选的质询 / 响应步骤。下面是由此产生的协议语法的形式（我修改了 NOM 来做到这一点）：

```
secure-nom      = open-peering *use-peering

open-peering    = C:OHAI *( S:ORLY C:YARLY ) ( S:OHAI-OK / S:WTF )

ORLY            = 1*mechanism challenge
mechanism       = string
challenge       = *OCTET

YARLY           = mechanism response
response        = *OCTET
```

其中 ORLY 和 YARLY 每个都包含一个字符串（在 ORLY 中是一个机制的列表，在 YARLY 中是一种机制）和不透明的二进制大对象数据。根据该机制，来自服务器的最初的质询可能是空的。我们不在乎一个摘记：我们只需把这个传给安全库来处理。

SASL RFC（*http://tools.ietf.org/html/rfc4422*）将详细介绍其他（我们不需要的）功能，以及 SASL 可以被攻击的各种方式，等等。

大型文件发布：FileMQ

让我们将所有这些技术汇集成一个文件分发系统，我把它叫作 FileMQ。这将是一个托管在 GitHub 上的真正的产品。我们在这里做的是第一个版本的 FileMQ，把它作为培训工具。如果这个概念能够工作，那么真正的东西可能最终会有一本专门讲述它的书。

412 为什么要制作 FileMQ

为什么要制作一个文件发布系统呢？我已经解释了如何通过 ØMQ 发送大文件，而且它真的很简单。但是，如果你想使能够利用消息传递的人数是可以使用 ØMQ 的人数的一百万倍，那么你需要另一种 API。一种我 5 岁的儿子也能理解的 API。一个普遍的、无须编程并能与几乎每一个单独的应用程序联用的 API。

是的，我说的是文件系统。这是 DropBox 模式：在某个地方扔下你的文件，当网络再次连接上时，它们被奇迹般地复制到别的地方。

不过，我的目标是一个完全分散式架构，看起来更像是 Git，这并不需要任何的云服务（尽管我们可以把 FileMQ 放在云中），而且也能执行多播（即可以一次性将文件发送到许多地方）。

FileMQ 必须是（能够实现）安全的，必须可以很容易地挂接到任意的脚本语言中，并且必须在我们的国内和办公网络中尽可能快地运行。

我想通过 WiFi 用它把照片从我的手机备份到我的笔记本电脑。在会议中对 50 台笔记本电脑实时共享演示幻灯片。与同事在会议中共享文档。从传感器把地震数据发送到中心集群。当我在抗议或骚乱期间拍摄视频时，从我的手机备份它。通过跨 Linux 服务器云来同步配置文件。

一个有远见的想法，是不是？嗯，创意不值钱。难在把这个创意做出来，并使其简单易用。

最初的设计切片：API

下面是我看到的第一个设计采用的方式。FileMQ 必须是分布式的，所以每个节点都可以同时是一个服务器和一个客户端。但我不希望协议是对称的，因为这似乎是被迫的。我们有从 A 点到 B 点的文件的自然流，其中 A 是“服务器”，B 是“客户端”，如果文件流回另一条路，我们就有两个流。FileMQ 还不是一个目录同步协议，但我们会使它相当接近。

因此，我要将 FileMQ 构建为两部分：客户端和服务器。然后，把这些一起放在可以同时充当客户端和服务器的主应用程序（filemq 工具）中。两个部件将看起来与 nom_server 非常相似，带有同类的 API：

```
fmq_server_t *server = fmq_server_new ();
fmq_server_bind (server, "tcp://*:5670");
fmq_server_publish (server, "/home/ph/filemq/share", "/public");
fmq_server_publish (server, "/home/ph/photos/stream", "/photostream");

fmq_client_t *client = fmq_client_new ();
fmq_client_connect (client, "tcp://pieter.filemq.org:5670");
fmq_client_subscribe (server, "/public/", "/home/ph/filemq/share");
fmq_client_subscribe (server,"/photostream/", "/home/ph/photos/stream");

while (!zctx_interrupted)                                          413
    sleep (1);

fmq_server_destroy (&server);
fmq_client_destroy (&client);
```

如果我们将这个 C API 包装到其他语言中，可以很容易地用脚本编写 FileMQ，并将其嵌入应用程序中，将它移植到智能手机，等等。

最初的设计切片：协议

协议的全名是文件消息队列协议，或 FILEMQ（大写，以便与软件区分开来）。首先，我们用 ABNF 语法编写协议。我们的语法从客户端和服务器之间的命令流开始。你应该将这些识别为我们已经看到的各种技术的组合：

```
filemq-protocol = open-peering *use-peering [ close-peering ]

open-peering    = C:OHAI *( S:ORLY C:YARLY ) ( S:OHAI-OK / error )

use-peering     = C:ICANHAZ ( S:ICANHAZ-OK / error )
                / C:NOM
                / S:CHEEZBURGER
                / C:HUGZ S:HUGZ-OK
                / S:HUGZ C:HUGZ-OK

close-peering   = C:KTHXBAI / S:KTHXBAI

error           = S:SRSLY / S:RTFM
```

下面是发往服务器和从服务器发出的命令：

```
;    客户端打开对服务器的探查
OHAI            = signature %x01 protocol version
signature       = %xAA %xA3
protocol        = string        ; 必须是 "FILEMQ"
string          = size *VCHAR
size            = OCTET
version         = %x01

;    服务器使用 SASL 模型质询客户端
ORLY            = signature %x02 mechanisms challenge
mechanisms      = size 1*mechanism
mechanism       = string
challenge       = *OCTET        ; ØMQ 帧

;    客户端用 SASL 认证信息来响应
YARLY           = %signature x03 mechanism response
response        = *OCTET        ; ØMQ 帧

;    服务器授予客户端访问权限
414 OHAI-OK     = signature %x04

;    客户端订阅到一个虚拟路径
ICANHAZ         = signature %x05 path options cache
path            = string        ; 全路径或路径前缀
options         = dictionary
dictionary      = size *key-value
key-value       = string        ; 格式化为名称=值
cache           = dictionary    ; 文件 SHA-1 签名

;    服务器确认订阅
ICANHAZ-OK      = signature %x06

;    客户端发送信用给服务器
NOM             = signature %x07 credit
credit          = 8OCTET        ; 64 位整数，网络顺序
sequence        = 8OCTET        ; 64 位整数，网络顺序

;    服务器发送一个文件数据块
CHEEZBURGER     = signature %x08 sequence operation filename
                  offset headers chunk
sequence        = 8OCTET        ; 64 位整数，网络顺序
operation       = OCTET
filename        = string
offset          = 8OCTET        ; 64 位整数，网络顺序
headers         = dictionary
```

```
chunk              = FRAME

;    客户端或服务器发送一个检测信号
HUGZ               = signature %x09

;    客户端或服务器响应一个信号检测
HUGZ-OK            = signature %x0A

;    客户端关闭探查
KTHXBAI            = signature %x0B
```

而下面是服务器告诉客户端某些东西出错的其他方式：

```
;    服务器错误应答 - 由于无访问权限而拒绝
S:SRSLY            = signature %x80 reason

;    服务器错误应答 - 客户端发送一个无效命令
S:RTFM             = signature %x81 reason
```

FILEMQ 驻留在 ØMQ unprotocols 网站（*http://rfc.zeromq.org/spec:19*），并带有一个 IANA（互联网编号分配机构）注册的 TCP 端口，端口号是 5670。

构建和尝试 FileMQ

415

FileMQ 软件栈驻留在 GitHub（*https://github.com/hintjens/filemq*）上。它的工作方式就像一个经典的 C/C++ 项目：

```
git clone git://github.com/hintjens/filemq.git
cd filemq
./autogen.sh
./configure
make check
```

你希望对这个软件使用最新的 CZMQ 主版本。现在试着运行 *track* 命令，它是一种简单的工具，使用 FileMQ 来跟踪一个目录的变化，并在另一个目录中做出同样的变化：

```
cd src
./track ./fmqroot/send ./fmqroot/recv
```

打开两个文件导航窗口，一个窗口进到 *src/fmqroot/send*，另一个窗口进到 *src/fmqroot/recv*。将文件拖放到 *send* 文件夹，你会看到它们也在 *recv* 文件夹中出现。服务器每秒检查一次新文件。在 *send* 文件夹中删除文件，而它们在 *recv* 文件夹中同时被删除。

我用 *track* 来更新我的 MP3 播放器之类的东西，将它们作为一个 USB 驱动器来安装。当我在笔记本电脑的 *Music* 文件夹中添加或删除文件时，在 MP3 播放器中发生同样的变

化。FILEMQ 还不是一个完整的复制协议，但我们将在以后解决这个问题。

内部架构

为了打造 FileMQ，我用了很多代码生成器，这对于一个教程可能是太多了。但是，代码生成器是可在所有其他软件栈中重复使用的，并对我们将在第 8 章完成的最终项目非常重要，它们是我们前面看到的一系列内容的演变：

- *codec_c.gsl* 生成一个给定协议的消息编解码器。
- *server_c.gsl* 生成协议和状态机的服务器类。
- *client_c.gsl* 生成协议和状态机的客户端类。

要学会利用 GSL 代码生成器的最佳方式是把这些翻译成你所选择的语言，并制作自己的示范协议和协议栈。你会发现它相当容易。FileMQ 本身并不试图支持多种语言。它可以，但它不会做不必要的复杂事情。

FileMQ 架构实际上切片成两层。有一组通用的类来处理块、目录、文件、补丁程序、SASL 安全和配置文件。然后，还有生成的软件栈：消息、客户端和服务器。如果创建一个新的项目，我会派生整个 FileMQ 项目，然后到那里修改如下三个模型：

416 - *fmq_msg.xml*，它定义了消息格式。
- *fmq_client.xml*，它定义了客户端状态机、API 和实现。
- *fmq_server.xml*，它定义了服务器状态机、API 和实现。

你可能想要重命名一些东西，以避免混淆。为什么我没有把可重复使用的类放到一个单独的库中呢？答案是双重的。首先，（还）没有人真正需要这个。其次，当你构建和使用 FileMQ 时，它会使你的东西更复杂。为解决一个理论问题而加入复杂性，这永远是不值得的。

虽然我用 C 语言编写 FileMQ，但它可以很容易地映射到其他语言。这是相当惊人的，当你添加 CZMQ 的通用 `zlist` 和 `zhash` 容器和类样式时，C 变得有多么美好。下面让我们快速浏览这些类：

- `fmq_sasl` 编码和解码一个 SASL 质询。我只实现了 PLAIN 机制，这也足以证明这个概念。
- `fmq_chunk` 适用于大小可变的二进制大对象数据。这些不与 ØMQ 的消息一样有效率，但它们做更少的怪事，因此更容易理解。该块类有方法来从磁盘读取数据块和写入数据块到磁盘。
- `fmq_file` 适用于文件，这可能在磁盘上存在，也可能不在磁盘上存在。它给你一个

文件的信息（如大小），并让你读取和写入文件、删除文件、检查文件是否存在，并检查文件是否是“稳定”的（稍后详述）。

- fmq_dir 适用于目录，从磁盘读取它们并比较两个目录，看看有什么变化。当有变化时，它返回一个“补丁”列表。
- fmq_patch 适用于一个补丁，这里的补丁实际上只是说：“创建这个文件”或“删除这个文件”（每次都指向一个 fmq_file 条目）。
- fmq_config 适用于配置数据。我随后就会介绍客户端和服务器的配置。

每个类都有一个测试方法，主要开发周期是“编辑，测试”，虽然这些大多是简单的自测，但它们造成我可以信任的代码和我知道仍然会导致破坏的代码之间的区别。这是一个安全的赌注，未被测试用例覆盖的任何代码都会有未被发现的错误。我不是外部测试工具的狂热爱好者，但当你编写功能时编写你的内部测试代码 ...... 这就像一把刀的手柄。

你真的应该能看懂源代码，并迅速了解这些类在做什么。如果你不能高兴地阅读这些代码，请告诉我。如果你想将这个 FileMQ 实现移植到其他语言，通过派生整个存储库开始，随后我们将看到，是否有可能在一个整体存储库中做到这一点。

公共 API

公共 API 包含两个类（如我们前面勾勒的）：

- fmq_client 提供了客户端 API，带有连接到一台服务器、配置客户端，以及注册路径的方法。 417
- fmq_server 提供了服务器 API，带有绑定到一个端口、配置服务器，以及发布一个路径的方法。

这些类提供了多线程的 API，我们已经用过几次的模型。当你创建一个 API 实例（即 fmq_server_new() 或 fmq_client_new()）时，这种方法揭开后台线程，做真正的工作，也就是运行在服务器或客户端。其他 API 方法然后再跟这个线程在 ØMQ 套接字（由在 inproc 上的两个 PAIR 套接字组成的一个“管道”）上交流。

如果我是一个急于在另一种语言中使用 FileMQ 的热心的年轻开发者，我可能会花费一个愉快的周末为这个公共 API 编写一个绑定，然后把它固定放在 *filemq* 项目中一个比如叫作“bindings/”的子目录，并发出提取请求。

实际的 API 方法出自状态机的描述，像这样（对于服务器）：

```
<method name = "publish">
<argument name = "location" type = "string"/>
<argument name = "alias" type = "string" />
```

```
mount_t *mount = mount_new (location, alias);
zlist_append (self->mounts, mount);
</method>
```

这个描述被变成了下面这段代码：

```
void
fmq_server_publish (fmq_server_t *self, const char *location, const char *alias)
{
    assert (self);
    assert (location);
    assert (alias);
    zstr_sendm (self->pipe, "PUBLISH");
    zstr_sendfm (self->pipe, "%s", location);
    zstr_sendf (self->pipe, "%s", alias);
}
```

设计说明

FileMQ 最难做的部分不是实现协议，而是在内部保持正确的状态。FTP 或 HTTP 服务器本质上是无状态的，而一个发布 - 订阅服务器必须至少维护订阅。

所以，我会仔细检查一些设计方面的内容：

- 客户端通过是否缺乏来自服务器的检测信号（HUGZ）来检测服务器是否死机。然
418 后它通过发送 OHAI 重新启动它的对话。OHAI 没有超时时间，因为 ØMQ DEALER 套接字将对传出消息无限期地排队。
- 服务器会通过是否有客户端缺乏其对检测信号的响应（HUGZ-OK）来检测客户端是否死机。在这种情况下，它会删除该客户端的所有状态，包括其订阅。
- 客户端 API 在内存中保存订阅，并且当它已成功连接时重放这些订阅。这意味着调用者可以随时订阅（并且不关心连接和验证实际发生在何时）。
- 服务器和客户端使用虚拟路径，这非常像一个 HTTP 或 FTP 服务器。你发布一个或多个“挂载点”，每个挂载点都对应于服务器上的一个目录。如果你只有一个挂载点，这些目录映射到一些虚拟路径，例如，“/”。客户端然后订阅虚拟路径，而文件到达一个收件箱目录。我们不通过网络发送物理文件名。
- 有一些时机的问题：如果在客户端连接和订阅时服务器创建了挂载点，则订阅将无法连接到正确的挂载点。所以，我们的最后一件事是绑定服务器端口。
- 客户端可以在任何时候重新连接，如果客户端发送 OHAI，这标志任何先前对话的结束和一个新对话的开始。我可能有一天使订阅耐久，以让它们在断开连接时生存下去。重新连接后，该客户端软件栈重放调用者应用程序已经做出的任何订阅。

配置

我已经建立了几个大型的服务器产品，如在 20 世纪 90 年代末流行的 Xitami Web 服务器，以及 OpenAMQ 消息服务器（*http://www.openamq.org/*）。简单且明显地获取配置是使这些服务器使用起来有乐趣的一大原因。

我们通常的目标是解决许多问题：

- 将默认的配置文件随产品发货。
- 允许用户添加一个永远不会被覆盖的自定义配置文件。
- 允许用户通过命令行进行配置。

然后再将这些逐一地叠加在另一个上面，所以命令行设置覆盖自定义设置，而自定义设置覆盖默认设置。要正确地这样做，这可能需要大量的工作。对于 FileMQ，我已经采取了较为简单的策略：所有的配置都是用 API 来完成的。

下面是我们启动和配置服务器的方法，例如：

```
server = fmq_server_new ();
fmq_server_configure (server, "server_test.cfg");
fmq_server_publish (server, "./fmqroot/send", "/");
fmq_server_publish (server, "./fmqroot/logs", "/logs");
fmq_server_bind (server, "tcp://*:5670");
```

我们使用特定格式的配置文件——ZeroMQ 属性语言（ZPL）（*http://rfc.zeromq.org/spec:4*）。我们几年前开始对 ØMQ“设备”使用这种简约的语法，但它对于任何服务器效果都很好： 419

```
#   配置服务器用于常规访问
#
server
    monitor = 1             #   检查挂载点
    heartbeat = 1           #   对客户端的信号检测

publish
    location = ./fmqroot/logs
    virtual = /logs

security
    echo = I: use guest/guest to login to server
    #   这些是我们接受的 SASL 机制
    anonymous = 0
    plain = 1
```

```
account
    login = guest
    password = guest
    group = guest
account
    login = super
    password = secret
    group = admin
```

生成的服务器代码做的一件可爱的事情（这似乎很有用）是解析这个配置文件（当你使用 fmq_server_configure() 方法），并执行匹配 API 方法的任何部分。因此，“publish”部分可以充当一个 fmq_server_publish() 方法。

文件稳定性

轮询一个目录的变化，然后对新文件做一些“有趣”的事，这是很常见的。但当一个进程正在写入文件时，其他进程是不知道该文件什么时候被完全写入的。一种解决办法是当我们创建第一个文件后，补充创建第二个“指示器”文件。但这是侵入性的。

还有一种更简洁的方式，这就是检测何时一个文件是“稳定”的（即，再也没有人写入它）。FileMQ 通过检查文件的修改时间做到这一点。如果它比当前时间早 1 秒以上，那么该文件被认为是稳定的，至少，对于被发到客户端来说足够稳定。如果一个进程在 5 分钟后过来并追加该文件，那么它会被再次发送。

要使这正常工作，这是任何希望成功使用 FileMQ 的应用程序要求的，请不要在内存中缓冲超过相当于 1 秒的数据。如果你使用的块非常大，文件可能在它不稳定时看起来稳定。

420

递交通知

我们正在使用的多线程的 API 模型的好处之一是，它本质上是基于消息的。这使得它非常适合把事件返回给调用者。一个更传统的 API 方法是使用回调函数，但跨越线程边界的回调函数有些微妙。下面是客户端在它已经收到了完整的文件时发回的一个消息：

```
zstr_sendm (self->pipe, "DELIVER");
zstr_sendm (self->pipe, filename);
zstr_sendf (self->pipe, "%s/%s", inbox, filename);
```

现在，我们可以把 _recv() 方法添加到等待从客户端返回事件的 API 中。它为调用者创造了一个整洁的风格：创建客户端对象、对其进行配置，然后接收并处理返回的任何事件。

符号链接

虽然简单的 API 使用一个临时区域是一个不错的选择，但它也对发送者产生了开销。如果我在一台摄像机上已经有了一个 2GB 大小的视频文件，我想把它通过 FileMQ 发送，目前的实现要求是，在把它发送给订阅者之前，要先将它复制到一个临时区域。

一种选择是挂载整个内容目录（例如，*/home/me/Movies*），但这是脆弱的，因为这意味着应用程序无法决定发送单个文件。这要么是全部发送，要么是一个也不发送。

一个简单的答案是实现可移植的符号链接。正如维基百科（*http://en.wikipedia.org/wiki/Symbolic_link*）上的解释：

> 符号链接包含一个自动解释和随后被操作系统当作一个路径指到另一个文件或目录的文本字符串。这个其他的文件或目录被称为“目标”。符号链接是独立于它的目标存在的第二个文件。如果一个符号链接被删除，它的目标不受影响。

这不会以任何方式影响协议，它是在服务器实现上的优化。让我们做一个简单的可移植的实现：

- 一个符号链接包含一个扩展名为 .ln 的文件名。
- 该文件名去掉 .ln 就是已发布的文件名。
- 链接文件包含一行，这是真正的文件路径。

因为我们已经在一个类（`fmq_file`）中收集了文件的所有操作，这是一个整洁的更改。当我们创建一个新的文件对象时，我们要检查它是否是一个符号链接，如果它是符号链接，那么所有只读操作（获取文件大小、读取文件）都针对目标文件，而不是该链接执行。

恢复和后期加入者 421

目前的情况是，FileMQ 有一个主要遗留问题：它没有为客户提供从故障中恢复的方法。该场景是一个客户端，连接到一台服务器，开始接收文件，然后由于一些原因断开连接。网络可能太慢，或中断。该客户端可能是在一台笔记本电脑，它被关闭然后又恢复。无线网络可能是断开连接的。当我们进入到一个更加移动的世界时（参见第 8 章），这个用例变得越来越频繁。在某些方面，它正在成为一个占主导地位的用例。

在经典 ØMQ 发布 - 订阅模式中，有两个强的基本假设，这两个假设在 FileMQ 的实际世界中通常都是错误的：第一个假设，数据非常迅速地失效，所以请求旧数据是没有意义的；第二个假设，网络是稳定的，很少中断（所以最好还是更多地投资于改善基础设施，少投资于解决恢复问题上）。

拿任何一个 FileMQ 用例来研究，你都会看到，如果客户端断开连接，并重新连接，它应该得到任何它错过了的东西。进一步的改进将是从局部故障中恢复，就像 HTTP 和 FTP 所做的那样。但让我们一次一件事地来做。

恢复的一个答案是“长期订阅”。FILEMQ 协议的第一稿的目标就是通过服务器可以保持和存储的客户端标识符来支持这样做，这样，如果客户端在出现故障后重新出现，则服务器就会知道哪些文件没有收到。

但是有状态的服务器制作起来很讨厌，且不易扩展。例如，我们如何才能将故障转移到辅助服务器呢？它从哪里得到它的订阅呢？如果每个客户端连接都独立工作，并承载所有必要的状态，这是好得多的办法。

长期订阅的另一个致命因素是，这种方法需要前期协调。前期协调始终是一个红色的标志，无论是对于在一个团队中一起工作的人，还是一堆互相交流的进程。那么后期加入者怎么办呢？在现实世界中，客户端并不整齐地排队，所有的人在同一时间说：“就绪！”在现实世界中，他们来去随意，如果我们能以对待全新客户端的同样方式对待已经消失然后回来的客户端，这是有价值的。

为了解决这个问题，我将在协议中添加两个概念：重新同步（resync）选项和缓存字段（字典）。如果客户想要恢复，它就设置重新同步选项，并通过缓存字段告诉服务器哪些文件它已经有了。两者我们都需要，因为在协议中没有办法来区分空的字段和一个空（null）字段。FILEMQ RFC 用如下文本描述了这些字段。

> “选项”（options）字段提供了额外的信息到服务器。服务器应该执行这些选项：
>
> 422 • RESYNC=1：如果客户端设置此项，服务器应当将虚拟路径的完整内容发送到客户端，除客户端已经有的文件外，文件用“缓存”字段中它们的 SHA-1 摘要来确定。
>
> 当客户端指定了 RESYNC 选项时，“缓存”字典字段告诉服务器客户端已经有了哪些文件。在“缓存”字典中的每个条目都是一个“文件名 = 摘要”键 / 值对，其中摘要应当是可打印的十六进制数据格式的 SHA-1 摘要。如果文件名以“/”开始，那么它应该从路径开始，否则服务器必须忽略它。如果文件名不是以“/”开始，那么服务器应当把它当作相对路径。

知道自己在经典的发布 - 订阅用例中的客户端不提供任何缓存数据，而希望恢复的客户端提供其缓存数据。它不需要在服务器中有状态，没有前期的协调，并同样适用于全新客户端（它可能已经通过一些带外手段收到了文件）和已经收到了一些文件，然后被断开了一段时间的客户端。

我决定使用 SHA-1 摘要有下面几个原因。首先，SHA-1 的速度足够快：在我的笔记本电脑上，对一个 25MB 的核心转储执行摘要需要 150 毫秒。其次，它是可靠的：不同版本的一个文件得到相同的散列值的机会足够接近零。第三，它是得到最广泛支持的摘要算法。循环冗余校验(如 CRC-32)虽然速度较快,但并不可靠。最近 SHA 版本(SHA-256、SHA-512）是更加安全的，但需要多花 50%以上的 CPU 周期，并且对于我们的需要是大材小用的。

当我们同时使用缓存和重新同步（这是从生成的编解码器类的 dump 方法输出的），一个典型的 ICANHAZ 消息的外形如下所示：

```
ICANHAZ:
    path='/photos'
    options={
        RESYNC=1
    }
    cache={
        DSCF0001.jpg=1FABCD4259140ACA99E991E7ADD2034AC57D341D
        DSCF0006.jpg=01267C7641C5A22F2F4B0174FFB0C94DC59866F6
        DSCF0005.jpg=698E88C05B5C280E75C055444227FEA6FB60E564
        DSCF0004.jpg=F0149101DD6FEC13238E6FD9CA2F2AC62829CBD0
        DSCF0003.jpg=4A49F25E2030B60134F109ABD0AD9642C8577441
        DSCF0002.jpg=F84E4D69D854D4BF94B5873132F9892C8B5FA94E
    }
```

虽然我们没有在 FileMQ 中做到这一点，但服务器可以使用缓存信息来帮助客户跟上它已经错过的删除操作。要做到这一点，就必须记录删除操作日志，然后当客户端订阅时，将这个日志与客户端缓存进行比较。

测试用例：曲目工具 423

要正确测试类似 FileMQ 的东西，我们需要一个使用实时数据的测试用例。我的一个系统管理员的任务是管理我的音乐播放器上的 MP3 曲目。顺便说一下，也就是用 Rock Box 重新整理的 Sansa 的剪辑，这是我强烈推荐的。当我下载曲目到我的 *Music* 文件夹时，我想将这些曲目复制到我的播放器，并且当我发现使我厌烦的曲目时，我在 *Music* 文件夹中删除它们，并希望那些文件也从我的播放器中删除。

我可以使用 bash 或 Perl 脚本编写这个用例——功能强大的文件分发协议干这个有点大材小用，但说实话，FileMQ 中最难的工作是目录比较的代码，并且我想从中受益。所以，我把这些汇集到一个叫“track”的调用 FileMQ API 的简单工具中。从命令行提供发送和接收的目录这两个参数来运行它：

```
./track /home/ph/Music /media/3230-6364/MUSIC
```

该代码是一个如何使用 FileMQ API 来完成本地文件分发的整洁的例子。下面是完整的程序，删去了许可证文本（它是 MIT/X11 许可）：

```
#include "czmq.h"
#include "../include/fmq.h"

int main (int argc, char *argv [])
{
    fmq_server_t *server = fmq_server_new ();
    fmq_server_configure (server, "anonymous.cfg");
    fmq_server_publish (server, argv [1], "/");
    fmq_server_set_anonymous (server, true);
    fmq_server_bind (server, "tcp://*:5670");

    fmq_client_t *client = fmq_client_new ();
    fmq_client_connect (client, "tcp://localhost:5670");
    fmq_client_set_inbox (client, argv [2]);
    fmq_client_set_resync (client, true);
    fmq_client_subscribe (client, "/");

    while (true) {
        //  从 fmq_client API 获取消息
        zmsg_t *msg = fmq_client_recv (client);
        if (!msg)
            break;              //  中断
        char *command = zmsg_popstr (msg);
        if (streq (command, "DELIVER")) {
            char *filename = zmsg_popstr (msg);
            char *fullname = zmsg_popstr (msg);
            printf ("I: received %s (%s)\n", filename, fullname);
            free (filename);
            free (fullname);
        }
        free (command);
        zmsg_destroy (&msg);
424 }
    fmq_server_destroy (&server);
    fmq_client_destroy (&client);
    return 0;
}
```

注意，我们如何在这个工具中使用物理路径。服务器发布的是物理路径“/home/ph/Music”，这映射到虚拟路径“/”。客户端订阅“/”并接收“/media/3230-6364/MUSIC”

中的所有文件。我可以在服务器目录中使用任何结构，它会被忠实地复制到客户端的收件箱。注意，API 方法 `fmq_client_set_resync()` 会导致一个服务器到客户端的同步。

得到一个官方端口号

我们在 FILEMQ 例子中一直使用端口 5670。不同于本书前面所有的例子，这个端口不是任意的，而是由互联网号码分配机构（Internet Assigned Numbers Authority，IANA）分配的，该机构负责 DNS 根、IP 寻址和其他互联网协议资源的全球协调。

我将很简要地解释何时以及如何为你的应用协议申请注册的端口号。这么做最主要的原因是为了确保你的应用程序可以在公共域中与其他协议不冲突地运行。从技术上讲，如果你交付使用 1024 和 49151 之间的端口号的任何软件，都应该只使用 IANA 注册的端口号。然而，很多产品不愿意做这种麻烦事，往往会转而使用 IANA 列表作为"要避免使用的端口"。

如果你的目标是制作有任何重要意义的公共协议，如 FILEMQ，你就会希望有一个 IANA 注册的端口。下面是完成这项工作的简要步骤：

- 明确地记录你的协议，因为 IANA 会需要你打算如何使用该端口的规范。虽然这不是一个正式的协议，但必须足够慎重，以通过专家评审。
- 决定你想使用什么传输协议：UDP、TCP、SCTP 等。通常对于 ØMQ，你会只想用 TCP。
- 在 *iana.org* 填写申请表，提供所有必要的信息。
- 然后，IANA 将通过电子邮件继续这个过程，直到你的申请被接受或拒绝为止。

请注意，如果你不要求特定端口号，IANA 会分配给你一个。因此，明智的做法是，在你交付软件前，而不是事后才开始这个申请过程。

第 8 章 425

分布式计算的框架

我们已经在许多方面认识了 ØMQ。现在，你可能已经开始使用我解释过的技术和其他你自己想通了的技术来建立自己的产品。你会开始面对如何使这些产品在现实世界中工作的问题。

但是，什么是“现实世界”？我会争辩说，世界正在成为移动部件日益增多的世界。有些人用短语“物联网”，暗示我们很快就会看到一个数量更大的新的设备类别，但它也比我们目前的智能手机、平板电脑、笔记本电脑和服务器更傻。不过，我完全不认为数据点是这样的。是的，是有越来越多的设备，但它们根本不傻。它们既聪明又强大，而且不断地变得更聪明更强大。

发挥作用的机制是一种我称之为“成本引力”的东西，它有每 18 ～ 24 个月降低技术成本一半的效果。换句话说，我们全球的计算能力每两年翻一番。未来充斥了万亿个强大的全功能多核计算机设备：它们不运行精简的“针对物品的操作系统”，而是全功能的操作系统和完整的应用程序。

这也是我们正用 ØMQ 瞄准的世界。当谈论“规模”时，我们不是指几百台电脑，也不是指几千台。将云视为微小、聪明，也许围绕每一个人自我复制的机器，它们填充每一个空间，覆盖每一面墙，填充裂缝，并最终成为了我们的一大部分，我们在出生前就得到它们，它们也跟着我们走向死亡。

这些微型机器的云不断地通过使用因特网协议（IP）的短距离无线链路互相交流。它们创建了网状网络，像神经信号一样传递信息和任务。它们增加了我们的记忆、我们的愿景、我们的通信的各个方面，和我们的身体功能。并且正是 ØMQ 为它们的对话和事件以及工作和信息交流提供动力。

现在，即便是为了让这一愿景的浅薄的模仿能在今天成为现实，我们也需要解决一系列 426

技术问题。这些问题包括：节点如何发现对方？它们如何跟现有的网络，比如 Web 进行交流？它们如何保护它们承载的信息？我们如何跟踪和监控它们，从而对它们在做什么获得一些了解？然后，我们需要做的是大多数工程师忘记的事情：将这个解决方案打包到一个让普通开发者非常容易使用的框架中。

这就是我们将尝试在本章做的事：构建分布式应用程序的框架，将它作为一个 API、协议和实现。这是一个不小的挑战，但我经常声称 ØMQ 使得这样的问题变得简单，所以让我们看看事实是否如此。

我们将讨论：

- 分布式计算的需求
- 用于邻近网络的无线网络的利弊
- 使用 UDP 和 TCP 探测
- 基于消息的 API
- 创建一个新的开源项目
- 对等网络连接（和谐模式）
- 跟踪节点的存在和消失
- 没有中央协调的消息群发
- 大规模测试和模拟
- 处理高水位标记和阻塞节点
- 分布式日志记录和监控

用于现实世界的设计

无论我们是通过 WiFi 连接一屋子移动设备还是通过模拟以太网连接一个虚拟机集群，都会碰到同类的问题。它们分别是：

发现

我们如何了解网络上的其他节点呢？我们是使用一个搜索服务、集中的中介，还是某种广播信标呢？

存在性

当其他节点来来去去时，我们如何跟踪它们呢？我们是使用某种集中注册服务，还是信号检测或信标呢？

连通性

我们到底怎么将一个节点连接到另一个节点呢？我们是使用本地网络、广域网络，

还是使用集中消息代理执行转发呢？

点对点的消息传递

我们如何将一个消息从一个节点发送到另一个节点呢？我们是把这个消息发送到节点的网络地址，还是通过一个集中的消息代理使用某个间接的寻址呢？ 427

消息群发

我们如何将一个消息从一个节点发送到一组其他节点呢？我们是通过一个集中的消息代理来工作，还是使用一种类似 ØMQ 的发布 - 订阅模型来工作呢？

测试和模拟

我们如何模拟大量的节点，以便我们可以适当地测试性能呢？我们必须买两打 Android 平板电脑，或者我们可以用纯软件来模拟吗？

分布式日志记录

我们如何跟踪这个节点云正在做什么，以便我们能发现性能问题和故障呢？我们是要创建一个主日志服务，还是允许每台设备来记录它周围的世界呢？

内容分发

我们如何将内容从一个节点发送到另一个节点呢？我们是使用诸如 FTP 或 HTTP 的以服务器为中心的协议，还是使用诸如 FileMQ 的分散的协议呢？

如果我们能够相当好地解决这些问题，那么进一步的问题就会出现（如安全性和广域桥接），我们会得到类似一个框架的东西，我将之称为“真酷的分布式应用程序”，而我的孙辈们可能把它叫作“我们的世界在上面运行的软件”。

你应该已经从我的反问中猜到了，我们有两个大方向可走。一是集中一切；另一种是分布一切。我要把赌注压在分布上。如果你想集中，你并不真的需要 ØMQ，你还可以使用其他选项。

所以，很不客气，下面是这个故事。一、随着时间的推移，移动部件数量成倍增加（每 24 个月翻倍）。二、这些部件停止使用电线，因为到处都是拖线变得非常烦人。三、未来的应用程序在这些部件的集群上用第 6 章介绍的仁慈暴君模式运行。四、今天构建这样的应用程序真的很难，不仅如此，还相当不可能。五、让我们用所有我们已经建立起来的技术和工具，使这能够廉价和容易地实现。六、庆祝！

无线网络的秘密生活

未来显然是无线的，而虽然现在许多大企业依靠将数据集中在他们的云中生存，但未来

看起来并不那么集中。每年，在我们网络边缘的设备都变得更聪明而不是更笨。它们渴望有工作和信息来处理并产生利润。它们不在周围拖电缆，除了每晚充一次电。所有这些
428 都是无线的，并且越来越多，它是按字母顺序排列的不同功能的 802.11 系列的无线网络。

为什么网状网络现在还没出现

作为未来世界如此重要的一个组成部分，无线网络有一个不被经常讨论的大问题，但依赖它的任何人需要对此做到心中有数。世界上的电话公司在几乎每一个政府发挥作用的国家都已经建立了自己不错且赚钱的手机卡特尔，这是基于说服政府，“如果没有对电波和想法的垄断权，这个世界就会崩溃”而做到的。从技术上讲，我们称之为“监管俘获”和“专利”，但实际上它是勒索和贪污腐败的一种形式。如果你（国家）给我（一个企业）乱要价、对市场收税，并且禁止所有真正的竞争对手开展活动的权利，我就给你 5%的提成。还不够吗？ 10%怎么样？好了，15%外加小费。如果你拒绝，我们就撤除服务。

而 WiFi 借用没有牌照的空域，并骑在开放和非专利并显著创新的互联网协议栈的背上，悄悄地绕过了这个。所以今天，我们会有如下的奇特情况，同样是从汉城打电话到布鲁塞尔，如果使用我们已经资助了几十年的国家支持的基础设施，我打一分钟电话要花好几个欧元，但如果我能找到一个不受管制的 WiFi 接入点，那一点钱都不用花。哦，以同样惊人的精确到零点零零的价格点（用任何你喜欢的货币），我还可以做视频、发送文件和照片，并下载整个家庭的电影所有这些事。上帝保佑，如果我尝试使用我实际支付的服务发送一张照片到我的家，这个花费将超过我买用来拍照的相机的价钱。

这是我们为容忍“相信我们，我们是专家”的专利制度这么久而付出的代价。但更重要的是，这对技术领域的大块头，特别是对拥有反互联网的 GSM、GPRS、3G 和 LTE 协议栈上的专利的芯片制造商，以及把电信运营商作为首要客户，积极扼杀无线网络的发展是一个巨大的经济诱因。

究其原因，对律师主导的“创新”的这种咆哮将引导你思考下面的问题，“如果 WiFi 真的是免费的会怎样？”这会发生在不太遥远的某一天，而这是值得下赌注的。我们会看到发生几件事情：第一，更加积极地利用空域，特别是对不存在干扰风险的近距离通信；第二，大容量的改进，当我们学习使用多个并行的空域时；第三，加速标准化的进程；而最后，在设备中对真正有意义的连接进行更广泛的支持。

现在，从你的手机将电影传输到电视被认为是“前沿技术”，这是荒谬的。让我们变得真正雄心勃勃。整个体育场的人一边观看比赛，一边实时互相分享照片和高清视频，最终用数码狂潮产生一个饱和空域的临时事件，怎么样？我应该能够在一个小时内从我身
429 边收集到 TB 级影像。为什么这个必须通过 Twitter 或 Facebook 和微小而昂贵的移动数据连接进行传输呢？家中数百台设备都通过网状网络互相交流，所以当有人按门铃时，

门廊的灯就会把视频传输到你的手机或电视，怎么样？一部小汽车可以跟你的手机交流并无须你插入电线就演奏你的电子舞曲播放列表，怎么样？

更加严重的是，为什么我们的数字社会要掌握在中央控制点的手中，他们监视、审查、记录、习惯于跟踪我们跟谁谈话，并针对我们的言论收集证据报告给当局，然后在当局认为我们有太多的言论自由时关闭这个数字社会呢？如果我们正在经历的隐私泄露只是单方面的，那它仅是一个问题，但这个问题随后将是灾难性的。一个真正的无线世界将绕过所有中央审查。互联网就是这么设计的，并且在技术上这是相当可行的。

一些物理知识

分布式软件的天真开发者将网络视为无限快和完全可靠的。虽然这对运行在以太网上的简单应用大致是真的，但无线网络迅速证明了神奇的思维和科学之间的区别。也就是说，无线网络在压力下如此容易受到极大的破坏，以至于我有时怀疑怎么会有人敢将它用于实际工作。当无线网络变得更好时，它的上限也向上移动，但从来没有足够快的速度来防止我们遇见它。

要了解无线网络在技术上如何执行，你需要了解物理学的基本定律：连接两个点所需的功率按照它们距离的平方增加。在更大的房子长大的人有成指数增大的嗓门，这是我在达拉斯学到的教训。对于 WiFi 网络，这意味着两个无线电设备被进一步分开时，它们必须要么使用更多的电力要么降低它们的信号速率。

在用户把设备视为无可救药的损坏之前，可以从一节电池里取出的只有这么多的电力。因此，即使一个 WiFi 网络可以被标以一个额定速度，接入点（AP）和客户端之间的实际位速率还要取决于两者相距多远。当你将具有 WiFi 功能的手机移到远离 AP 的地方时，试图与对方交流的这两个无线电设备会首先增加它们的耗电量并随后降低其比特率。

如果我们要构建可靠的分布式应用程序，而不像木偶那样在它们身后吊着钢丝，这个效应有下面一些我们应该意识到的后果：

- 如果你有一组设备与 AP 交流，那么当 AP 与最慢的设备交流时，整个网络都必须等待。这就像在一个聚会上给指定驾驶员讲一个笑话，如果他没有幽默感，仍然完全清醒，并且对语言把握不好，就必须对他重复讲。
- 如果使用单播 TCP 并发送消息到多个设备，AP 必须分别将数据包发送到每个设备，是的，你知道这一点，以太网的工作方式也是这样的。但现在要明白，一个遥远的（或低功率）设备将强制所有东西都在等待那个最慢的设备迎头赶上。
- 如果你使用多播或广播（在大多数情况下，这二者工作方式相同），则 AP 会一次性 430
把一个数据包发送到整个网络，这真棒，但它会用最慢的比特率发送（通常为 1Mb/s）。

在一些 AP 上，你可以手动调节这个速度，但这么做只会减少你的 AP 的覆盖范围。你也可以购买更智能一点，并会找出它们可以放心使用的最高比特率的更昂贵的 AP，或者你可以使用带有 Internet 组管理协议（IGMP）支持的企业级 AP 和 ØMQ 的 PGM 传输协议，只将数据包发送给订阅的客户端。但是，我永远不会把赌注压在这些 AP 会得到广泛的使用上。

当你试图把更多的设备连到一个 AP 时，性能就会迅速恶化到再增加一个设备就可能破坏整个网络的每一台设备的这种地步。许多 AP 通过当连接的设备数到达某个上限就随机断开客户端来解决这个问题，例如移动热点的上限为 4 ～ 8 台设备，消费级的 AP 为 30 ～ 50 台设备，而企业级 AP 为 100 台设备。

现状是什么

尽管 WiFi 作为不知何故逃脱到野外的企业技术，其角色是尴尬的，但它的用途已经超过免费的 Skype 电话。虽然它还不理想，但对于我们解决一些有趣的问题已经工作得足够好。让我给你提供一个快速的状态报告。

首先，点对点与 AP 到客户端的对比。传统的无线网络都是 AP 客户端。每一个数据包都必须从客户端 A 跑到 AP，然后到客户端 B，虽然你把你的带宽砍去了 50%，但这只是问题的一半。我在前面解释了有关反幂法则。如果 A 和 B 非常接近，但两者都远离接入点，那么它们都会使用一个低比特率。想象一下，你的 AP 是在车库里，而你在客厅试图从手机把视频传输至电视。祝你好运！

有一个古老的，让 A 和 B 相互交流的“特别”模式，但它对于任何有趣的东西都太慢，当然，它在所有的移动芯片组上都是被禁用的。实际上，它在芯片制造商友好地提供给硬件制造商的绝密的驱动程序中被禁用。还有一个新的隧道式直接链路建立（TDLS）协议，允许两个设备使用 AP 的发现但不把 AP 用于传输来建立直接的链路。并有一个“5G”WiFi 标准（这是一个营销术语，所以把它放在引号中），可将连接速度提高到千兆比特 / 秒。TDLS 和 5G 一起使得高清电影从手机传输到电视上成为一个虚晃的现实。我设想 TDLS 会被加以各种方式的限制，以安抚电信运营商。

此外，经过一个迅速的 10 年左右的工作，2012 年，我们看到了 802.11s 网状网络协议的标准化。网状网络完全消除接入点，至少在对它存在和被广泛使用的未来的想象中如此。
431 设备直接相互交流，并维护让它们转发数据包的邻居的小路由表。试想一下，AP 软件嵌入到每一个设备，但足够聪明（这并不像它听上去那样令人印象深刻）以执行多跳。

从移动数据敲诈勒索狂欢赚钱的人（译者注：指大电信运营商）都不希望看到 802.11s 可用，因为城市级的网状网络就是这样一个触及他们底线的噩梦，所以它的发生会尽可

能地慢。唯一有力量（和，我假设地对地导弹）推动网状网络技术广泛应用的大型组织是美国陆军。但网状网络将进一步显现，我敢打赌，到 2020 年左右，802.11s 将会被广泛使用在消费型电子产品中。

其次，如果我们没有点对点，在多远的距离内，我们才能相信今天的 AP 呢？好吧，如果你在美国去星巴克咖啡厅并尝试使用两台笔记本电脑通过免费无线网络连接来运行 ØMQ 的“Hello World”的例子，你会发现它们无法连接。为什么呢？答案在其名称“attwifi”中。AT & T 是一个很好的老旧现任电信运营商，它憎恨 WiFi 并可能廉价地为星巴克和其他人提供服务，以阻止独立运营商进入市场。但你购买的任何接入点其实都支持客户端 -AP- 客户端访问，并且在美国以外我从来没有发现一个公共接入点是用 AT & T 的方法锁定的。

第三，性能。AP 显然是一个瓶颈，即使你把 A 和 B 实际就放在 AP 的旁边，也不能得到比其广告宣传的一半更好的速度。更糟的是，如果在同一空域有其他的 AP，它们会互相干扰。在我家里，无线网络几乎根本不能工作，因为楼下两层的邻居有一个已经被他们放大的 AP。即使是在不同的频道，它也干扰了我家的 WiFi。在我现在所在的咖啡厅有超过一打的网络。实际上，只要我们依赖基于 AP 的无线网络，就会受到随机干扰而性能是不可预知的。

第四，电池的使用寿命。例如，无线网络在空闲时比蓝牙更饥渴并没有内在的原因。它们使用相同的无线电波和低级别的组帧。主要的区别是在调谐和协议中。为了使无线节电运行良好，设备必须要多休眠，并每隔一段时间只给其他设备发一次信标。要使这能正常工作，它们需要同步它们的时钟。对于手机部分，常常发生这种情况，这就是为什么我的旧的翻盖手机充一次电可以运行五天。而当无线网络工作时，它会使用更多的电能。目前的功率放大器技术也是低效的，这意味着你从你的电池获取的能量比你输出到空中的能量多了很多(浪费的电能变成热量致使手机发烫)。当人们更注重手机的 WiFi 时，功率放大器也在改善中。

最后，移动接入点。如果我们不能相信集中的接入点，并且如果我们的设备有足够的智能来运行完整的操作系统，何不让它们作为 AP 工作呢？我很高兴你问这个问题。是的，我们可以，并且它工作得相当不错。特别是因为我们在一个像 Android 的现代操作系统上可以在软件中开启和关闭这个功能。同样，这一块的“阻挡者”是美国电信运营商，他们非常讨厌这个功能并在他们所控制的手机中杀死或削弱它。聪明的电信公司意识到这是一种扩大他们的“最后一公里”的方式，并给更多的用户带来更高价值的产品，但运营商不在智慧上竞争。

432

结论

WiFi 不是以太网，虽然我相信未来 ØMQ 应用程序将有一个非常重要的分散式的无线存在，但这不会是一帆风顺的。你想指望从以太网获得的许多基本可靠性和容量都丢失了。当你通过 WiFi 运行分布式应用程序时，必须允许频繁超时、随机等待时间、任意断开连接、整个接口关闭和开启，等等。

对无线网络的技术演进最恰当的描述是“缓慢而无趣的”。试图利用分布式无线的应用程序和框架大多缺席或差劲。唯一现有的用于邻近网络的开源框架是来自高通的 AllJoyn（*https://www.alljoyn.ory*）。但使用 ØMQ 我们证明了现有的选手的惰性和老朽无能，没有理由让我们继续静坐下去了。当我们准确理解问题后，我们可以解决这些问题。我们可以把梦想变成现实。

发现

一个关于短距离无线的好东西是邻近。无线网络紧密地映射到物理空间，它紧密地映射到我们的自然组织方式上。事实上，互联网是相当抽象的，这混淆了很多“对它有点了解”，但实际上不了解的人。有了无线网络，我们就有了可能是超具体的技术性的连接。你看到你所得到的，同时你得到你所看到的。具体是指容易理解，而且这应该意味着来自用户的爱，而不是典型的挫折和沉默的沸腾的仇恨。

邻近是关键。假设在一个房间中有一堆 WiFi 无线模块，它们愉快地互发信标给对方。对于很多应用程序来说，它们可以找到对方，开始聊天，而无须任何用户输入是有道理的。毕竟，大多数现实世界的数据不是私人的，它只是非常局部的。

作为以 ØMQ 为基础的邻近网络的第一步，让我们来看看如何做到发现。虽然存在做这项工作的库，但我不喜欢它们。它们似乎过于复杂和过于具体，还以某种方式回溯到人们意识到分布式计算可以从根本上简单到之前的史前时代。

通过原始套接字先发制人的发现

我在首尔江南区的酒店房间中，具有 4G 无线热点，一台安装 Linux 的笔记本电脑，和两部 Android 手机。手机和笔记本电脑正与热点交流。*ifconfig* 命令显示我的 IP 地址为 192.168.1.2，让我尝试一些 *ping* 指令。动态主机控制协议（DHCP）服务器往往按顺序抛出地址，所以我的手机 IP 地址可能是靠近的，用数字来说话：

433

```
$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_req=1 ttl=64 time=376 ms
```

```
64 bytes from 192.168.1.1: icmp_req=2 ttl=64 time=358 ms
64 bytes from 192.168.1.1: icmp_req=4 ttl=64 time=167 ms
^C
--- 192.168.1.1 ping statistics ---
3 packets transmitted, 2 received, 33% packet loss, time 2001ms
rtt min/avg/max/mdev = 358.077/367.522/376.967/9.445 ms
```

发现一个！ 150 ~ 300 毫秒的往返延迟 …… 这是一个令人惊讶的大数字，要牢记此结果以备将来使用。现在我 *ping* 自己，只是为了尽量仔细地检查事情：

```
$ ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_req=1 ttl=64 time=0.054 ms
64 bytes from 192.168.1.2: icmp_req=2 ttl=64 time=0.055 ms
64 bytes from 192.168.1.2: icmp_req=3 ttl=64 time=0.061 ms
^C
--- 192.168.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.054/0.056/0.061/0.009 ms
```

现在，响应有点快了，这是我们所期望的。尝试接下来的几个地址：

```
$ ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_req=1 ttl=64 time=291 ms
64 bytes from 192.168.1.3: icmp_req=2 ttl=64 time=271 ms
64 bytes from 192.168.1.3: icmp_req=3 ttl=64 time=132 ms
^C
--- 192.168.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 132.781/231.914/291.851/70.609 ms
```

这是第二部手机，它与第一部具有相同种类的延迟。让我们继续，看看是否有任何其他设备连接到热点：

```
$ ping 192.168.1.4
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.
^C
--- 192.168.1.4 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2016ms
```

那就是它。现在，*ping* 使用原始 IP 套接字发送 ICMP_ECHO 消息。关于 ICMP_ECHO 有用的事情是，它从任何并没有刻意关闭回应的 IP 栈得到一个响应。对于害怕老旧的“Ping 到死”漏洞的企业网站，因为畸形的消息可能导致机器崩溃，所以这仍然是一种普遍做法。

我把这称为*先发制人的发现*，因为它并不需要从设备得到任何合作。我们不依靠来自手

机的任何合作来查看它们位于那里，只要它们不主动忽略我们，我们就可以看到它们。

434 你可能会问，为什么这是有用的。我们不知道，对 ICMP_ECHO 做出响应的节点是否运行 ØMQ，它们是否有兴趣和我们交流，它们有什么我们可以使用的服务，甚至它们的设备种类是什么。然而，知道有某个东西在地址 192.168.1.3 上已经很有用了。我们也知道设备有多远（相对），我们知道在网络上有多少设备，我们知道了网络的大致状态（如：好、差或极差）。

创建 ICMP_ECHO 消息并发送它们甚至不用付出太多努力。这只需要几十行代码，并且可以使用 ØMQ 多线程来对在我们自己的 IP 地址的上方和下方伸出的地址并行执行此操作。这可能是一种乐趣。

然而，可悲的是，我使用 ICMP_ECHO 来发现设备的想法有一个致命的缺陷。打开一个原始 IP 套接字，这需要在一台 POSIX 电脑上的 root 权限。这会阻止流氓程序获得针对其他人的数据。可以通过给我们的命令授予 *sudo* 权限（*ping* 有所谓的粘滞位设置）来获得打开 Linux 原始套接字的权力。但一个诸如 Android 的移动操作系统，它需要 root 访问权限（即破解手机或平板电脑的 root）。这个问题超出大多数人的能力，所以 ICMP_ECHO 对大多数设备是遥不可及的。

让我们尝试在用户空间做一些事。大多数人采取的后续步骤是 UDP 多播或广播。让我们沿着这条路走。

使用 UDP 广播协同发现

人们往往认为多播比广播更现代化和“更好”。在 IPv6 中，广播根本不能工作：你必须使用多播。尽管如此，无论如何，所有的 IPv4 的本地网络发现协议最终使用的都是 UDP 广播。究其原因：除了广播更简单，风险更小外，广播和多播的最终工作大致相同。多播被网络管理员视为是一种危险，因为它会跨网段渗透。

如果你从来没有用过 UDP，你会发现它是一个相当不错的协议。在某些方面，它让我想起了 ØMQ，它采用两种不同的方式将整个消息发送到对等节点：一对一和一对多。使用 UDP 的主要问题是：(a) POSIX 套接字 API 是专为通用的灵活性，而不是简单性设计的，(b) UDP 消息的实际用途限制在大约 512 字节，以及 (c) 当你开始使用 UDP 对真正的数据进行传输时，会发现有很多消息被丢弃，尤其在基础设施对 TCP 的偏好胜过 UDP 的时候。

示例 8-1 是使用 UDP，而不是 ICMP_ECHO 的一个最精简的 ping 程序。

示例8-1：UDP发现，模型1（udpping1.c）

```
//
//  UDP ping 命令
//  模型 1，内联做 UDP 工作
//
#include <czmq.h>                                                    435
#define PING_PORT_NUMBER 9999
#define PING_MSG_SIZE    1
#define PING_INTERVAL    1000  //  间隔为每秒一次

static void
derp (char *s)
{
    perror (s);
    exit (1);
}

int main (void)
{
    zctx_t *ctx = zctx_new ();

    //  创建 UDP 套接字
    int fd;
    if ((fd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
        derp ("socket");

    //  请求操作系统允许我们从套接字执行广播
    int on = 1;
    if (setsockopt (fd, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on)) == -1)
        derp ("setsockopt (SO_BROADCAST)");

    //  将 UDP 套接字绑定到本地端口，以便我们能接收 ping 命令
    struct sockaddr_in si_this = { 0 };
    si_this.sin_family = AF_INET;
    si_this.sin_port = htons (PING_PORT_NUMBER);
    si_this.sin_addr.s_addr = htonl (INADDR_ANY);
    if (bind (fd, &si_this, sizeof (si_this)) == -1)
        derp ("bind");

    byte buffer [PING_MSG_SIZE];
```

我们使用 `zmq_poll()` 等待 UDP 套接字上的活动，因为该函数适用于非 ØMQ 文件句柄。我们每秒发送一次信标，并且收集和报告从其他节点发来的信标。示例 8-2 显示了 *ping* 的主循环。

示例8-2：UDP发现，模型1（udpping1.c）：ping主循环

```
    zmq_pollitem_t pollitems [] = {{ NULL, fd, ZMQ_POLLIN, 0 }};
    //  立即发送第一个 ping 命令
    uint64_t ping_at = zclock_time ();

    while (!zctx_interrupted) {
        long timeout = (long) (ping_at - zclock_time ());
        if (timeout < 0)
            timeout = 0;
        if (zmq_poll (pollitems, 1, timeout * ZMQ_POLL_MSEC) == -1)
            break;              //  中断

        //  某个节点回应了我们的 ping 命令
        if (pollitems [0].revents & ZMQ_POLLIN) {
            struct sockaddr_in si_that;
            socklen_t si_len;
            ssize_t size = recvfrom (fd, buffer, PING_MSG_SIZE, 0, &si_that,
                    &si_len);
            if (size == -1)
                derp ("recvfrom");
            printf ("Found peer %s:%d\n",
                inet_ntoa (si_that.sin_addr), ntohs (si_that.sin_port));
        }
        if (zclock_time () >= ping_at) {
            //  广播信标
            puts ("Pinging peers...");
            buffer [0] = '!';
            struct sockaddr_in si_that = si_this;
            inet_aton ("255.255.255.255", &si_that.sin_addr);
            if (sendto (fd, buffer, PING_MSG_SIZE, 0, &si_that,
                    sizeof (struct sockaddr_in)) == -1)
                derp ("sendto");
            ping_at = zclock_time () + PING_INTERVAL;
        }
    }
    close (fd);
    zctx_destroy (&ctx);
    return 0;
}
```

436

此代码使用一个单独的套接字广播 1 字节的消息并接收其他节点广播的任何东西。当我运行它时，它仅显示一个节点，就是它本身：

```
Pinging peers...
Found peer 192.168.1.2:9999
```

```
Pinging peers...
Found peer 192.168.1.2:9999
```

如果我关掉所有的网络，然后再试一次，正如我期望的，现在发送消息失败了：

```
Pinging peers...
sendto: Network is unreachable
```

在解决目前瞄准你喉咙的问题的基础上工作，让我们来修复这第一个模型的最紧迫的问题。这些问题分别是：

- 使用 255.255.255.255 广播地址是有点可疑的。一方面，这个广播地址精确地是指"发送到本地网络上的所有节点，并且不转发。"不过，如果你有多个接口（有线以太网，WiFi），将只在你的默认路由上广播出去，并只通过一个接口。我们想要做的是，要么把我们的广播发送到每个接口的广播地址，要么找到 WiFi 接口和广播地址。
- 正如套接字编程的许多方面，获得网络接口的信息是不可移植的。我们希望在应用程序中写入不可移植的代码吗？不，最好是把它隐藏在一个库中。 437
- 除了"终止"，没有对错误的处理，这对瞬态问题，如"你的无线网络已关闭"是太粗暴了。代码应该区分软错误（忽略并重试）和硬错误（断言）。
- 代码需要知道自己的 IP 地址，并且忽略它发送出去的信标。正如寻找广播地址，这也需要检查可用的接口。

这些问题最简单的解决方案，是将 UDP 的代码推到一个单独的库中，它提供一个整洁的 API，就像下面这样：

```
//  构造函数
static udp_t *
    udp_new (int port_nbr);

//  析构函数
static void
    udp_destroy (udp_t **self_p);

//  返回 UDP 套接字句柄
static int
    udp_handle (udp_t *self);

//  使用 UDP 广播发送消息
static void
    udp_send (udp_t *self, byte *buffer, size_t length);

//  从 UDP 广播接收消息
static ssize_t
```

```
        udp_recv (udp_t *self, byte *buffer, size_t length);
```

示例 8-3 显示了调用该库的重构的 UDP ping 程序，它更整洁和更好。

示例8-3：UDP发现，模型2（udpping2.c）

```
//
//  UDP ping 命令
//  模式 2，使用单独的 UDP 库
//
#include <czmq.h>
#include "udplib.c"

#define PING_PORT_NUMBER 9999
#define PING_MSG_SIZE    1
#define PING_INTERVAL    1000  //  间隔为每秒一次

int main (void)
{
    zctx_t *ctx = zctx_new ();
438 udp_t *udp = udp_new (PING_PORT_NUMBER);

    byte buffer [PING_MSG_SIZE];
    zmq_pollitem_t pollitems [] = {{ NULL, udp_handle (udp), ZMQ_POLLIN, 0 }};

    //  立即发送第一个 ping 命令
    uint64_t ping_at = zclock_time ();

    while (!zctx_interrupted) {
        long timeout = (long) (ping_at - zclock_time ());
        if (timeout < 0)
            timeout = 0;
        if (zmq_poll (pollitems, 1, timeout * ZMQ_POLL_MSEC) == -1)
            break;              //  中断

        //  某个节点回应了我们的 ping 命令
        if (pollitems [0].revents & ZMQ_POLLIN)
            udp_recv (udp, buffer, PING_MSG_SIZE);

        if (zclock_time () >= ping_at) {
            puts ("Pinging peers...");
            buffer [0] = '!';
            udp_send (udp, buffer, PING_MSG_SIZE);
            ping_at = zclock_time () + PING_INTERVAL;
        }
    }
    udp_destroy (&udp);
```

```
    zctx_destroy (&ctx);
    return 0;
}
```

udplib 这个库隐藏了很多不愉快的代码（当我们让这工作在更多的系统上时，它们将变得更丑陋）。我不打算在这里打印代码，但你可以在存储库（*https://github.com/imatix/zguide/blob/master/examples/C/udplib.c*）中读取它。

现在我们估计有更多的问题并想知道它们是否可以把我们当午饭吃了。首先，IPv4 与 IPv6 的对比和多播与广播的对比。在 IPv6 中，广播根本不存在，人们使用多播。从我使用 WiFi 的经验看，IPv4 的多播和广播的工作相同，除了在某些广播正常工作而多播会损坏的情况下。问题是，一些无线接入点不转发多播报文。当你有一台充当移动 AP 的设备（如平板电脑）时，它可能也不会获得多播报文，这意味着它会看不到网络上的其他节点。

目前，最简单可行的解决方案是简单地忽略 IPv6，并使用广播。一个也许是更明智的解决方案是使用多播并处理不对称的信标，如果它们发生的话。

目前，我们将坚持保持愚蠢和简单。以后我们总有时间使其更加复杂的。

一台设备上的多个节点 439

因此，我们可以发现 WiFi 网络上的节点，只要它们如我们预期地发送信标。但是，当我尝试用两个进程进行测试这个时，运行 *udpping2* 两次，第二个实例抱怨“‘绑定上的’地址已在使用”，然后退出。哦，对了。如果你尝试把两种不同的套接字绑定到同一端口，UDP 和 TCP 都会返回一个错误。这是正确的，在一个套接字上，两个阅读器的语义将是不可思议的，至少可以这样说。它们分别读取奇 / 偶字节吗？是你获取所有的 1，而我获取所有的 0 吗？

然而，快速检查 *stackoverflow.com* 和回忆一下被称为 SO_REUSEADDR 的套接字选项的结果是很珍贵的。如果我使用那个选项，就可以把多个进程绑定到同一个 UDP 端口，而它们将接收所有到达该端口的任何消息。设计了这个东西的家伙仿佛猜透了我的心！（这种方式比我重新发明轮子更合理。）

快速测试显示，SO_REUSEADDR 如它承诺的那样工作。这是很好的，因为我想要做的下一件事情是设计一个 API，然后启动几十个节点，来看它们是否能发现对方。必须在一个单独的设备上测试每个节点，这将是非常麻烦的。而当我们开始测试在大的片状网络上实际流量的行为如何时，这两个备选方案是模拟或暂时混乱的。

而从我的经验来说：今年夏天，我们正在一次测试几十台设备。建立一套完整的测试并

使它运行需要一个小时左右，而且，如果你想要得到任何一种可重复的结果，都需要一个屏蔽无线干扰的空间。

如果我是一个高明的Android开发者并有空闲的周末，我会立刻（比如，这会花我两天时间）将此代码移植到我的手机，并用它发送信标到我的台式电脑。但有时懒惰更加有利可图。我喜欢我的Linux笔记本电脑。我喜欢能够从一个进程启动十几个线程，并使每个线程的行为都像一个独立的节点。在我可以在我的笔记本电脑上模拟一个的时候，我喜欢不必在一个实际的法拉第笼（译者注：一种静电屏蔽装置）中工作。

设计API

我要在一台设备上运行*N*个节点，而它们将不得不去发现对方，并且还在本地网络上发现了在那之外的一堆其他节点。我可以使用UDP进行本地发现以及远程发现。虽然比方说，可以认为这不如使用ØMQ `inproc`传输协议有效率，但它具有一个很大的优势，即完全相同的代码将同时工作在模拟和实际部署环境中。

如果在一台设备上有多个节点，我显然不能使用IP地址和端口号作为节点地址。我需要一些逻辑节点标识符。可以认为，该节点标识符仅需在该设备的上下文中是唯一的。我的脑子里充满可以做的复杂的东西，比如位于实际的UDP端口并给内部节点转发消息的超级节点。我以头撞桌，直到把发明新概念的想法从脑子中驱除。

440 经验告诉我们，在应用程序正在运行时，WiFi会做类似消失又重新出现的事。用户点击某个东西，产生比如在一个会话的中途更改IP地址的有趣结果。我们不能依赖于IP地址，也不能依赖已建立的连接（在TCP方式中）。我们需要某些长效解决机制，它可以在被拆掉，然后重新创建的接口和连接中生存。

下面是我能看到的最简单的解决方案：我们给每个节点一个UUID，并且指定那些由它们的UUID代表的节点，可以出现或重新出现在某些IP的“地址：端口”端点，然后再次消失。我们稍后会从丢失的消息执行恢复。一个UUID有16字节。所以，如果在一个WiFi网络上有100个节点，那么只是为了发现和存在，空中传播必须承载每秒钟3200字节（对其加倍用于其他随机的东西）的信标数据。这似乎是可以接受的。

回到概念。对于我们的API，我们确实需要一些名字。至少，我们需要一种方法来区分表示“我们”的节点对象和表示我们的对等节点的节点对象。我们会做如下的事情，创建一个“我们”，然后问它认识多少对等节点，以及它们分别是谁。术语“对等节点”是很清楚的。

但从开发者的角度来看，一个节点（应用程序）需要有一种方式来跟外面的世界交流。让我们借用一个来自网络的术语，并把这样的一个方式称为“接口”。接口向其他所有

节点描述我们，并向我们描述其他所有节点，把它们描述为一组其他的对等节点。它会自动执行任何它必须做的发现。当我们希望与某个对等节点交流时，让接口为我们做这件事。而当对等节点跟我们交流时，给我们提供信息的也正是接口。

这似乎是一个整洁的 API 设计。它的内部实现是怎么样的呢？

- 该接口必须是多线程的，因此，当前台的 API 与应用程序交流时，一个线程可以在后台执行 I/O。我们在克隆和自由职业者的客户端 API 中使用了这种设计。
- 接口后台线程做发现的业务：绑定到 UDP 端口，发送 UDP 信标，并接收信标。
- 我们至少需要在信标消息中发送 UUID，以使我们可以区分自己的信标和对等节点的信标。
- 我们需要跟踪出现和消失的对等节点。为此，我将使用一个散列表来存储所有已知的节点，并在某个超时时间后使对等节点过期。
- 我们需要一种方法来将对等节点和事件汇报给调用者。在这里，我们进入一个有趣的问题。一个后台 I/O 线程如何把这件工作正在发生的情况告诉前台线程 API 呢？也许能用回调函数来完成？哎呀，不行。当然，我们将使用 ØMQ 消息来做这件事。

示例 8-4 所示的是 UDP 的 *ping* 程序的第三次迭代，比第二个更简单，更漂亮。用 C 编写的主函数，只有 10 行代码。

示例8-4：UDP发现，模型3（udpping3.c） 441

```
//
//  UDP ping 命令
//  模型 3，采用抽象的网络接口
//
#include <czmq.h>
#include "interface.c"

int main (void)
{
    interface_t *interface = interface_new ();
    while (true) {
        zmsg_t *msg = interface_recv (interface);
        if (!msg)
            break;              //  中断
        zmsg_dump (msg);
    }
    interface_destroy (&interface);
    return 0;
}
```

如果你已经研究过我们如何制作多线程的 API 类，就应该熟悉该接口的代码（参见示例 8-5）。

示例8-5：UDP ping接口（interface.c）

```
//  接口类
//  这实现了一个针对我们的节点网络的“接口”

#include <czmq.h>
#include <uuid/uuid.h>
#include "udplib.c"


//  =====================================================================
//  同步部分，在我们的应用程序线程中工作

//  ---------------------------------------------------------------------
//  接口类的结构

typedef struct {
    zctx_t *ctx;                //  上下文包装器
    void *pipe;                 //  通向代理的管道
} interface_t;

//  这是处理我们实际接口类的线程
static void
    interface_agent (void *args, zctx_t *ctx, void *pipe);
```

442 示例 8-6 给出了接口类的构造函数和析构函数。请注意，这个类几乎没有任何属性，它只是一个启动后台线程的借口和一个围绕 zmsg_recv() 的包装器。

示例8-6：UDP ping接口（interface.c）：构造函数和析构函数

```
interface_t *
interface_new (void)
{
    interface_t
        *self;

    self = (interface_t *) zmalloc (sizeof (interface_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, interface_agent, NULL);
    return self;
}

void
interface_destroy (interface_t **self_p)
```

```
{
    assert (self_p);
    if (*self_p) {
        interface_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}
```

在示例 8-7 中，我们等待来自接口的消息。这将给我们返回一个 zmsg_t 对象，或如果中断，则返回 NULL。

示例8-7：UDP ping接口（interface.c）：接收消息

```
static zmsg_t *
interface_recv (interface_t *self)
{
    assert (self);
    zmsg_t *msg = zmsg_recv (self->pipe);
    return msg;
}


// =====================================================================
// 异步部分，在后台工作
```

示例 8-8 所示的结构中定义了我们要发现和跟踪的每个对等节点。

示例8-8：UDP ping接口（interface.c）：对等节点类

```
typedef struct {
    uuid_t uuid;                // 作为二进制大对象的对等节点的 UUID
    char *uuid_str;             // 作为可打印字符串的 UUID
    uint64_t expires_at;
} peer_t;

#define PING_PORT_NUMBER 9999
#define PING_INTERVAL    1000   // 每秒钟一次
#define PEER_EXPIRY      5000   // 5 秒钟后它就已经离去

// 将二进制 UUID 转换成新分配的字符串

static char *
s_uuid_str (uuid_t uuid)
{
    char hex_char [] = "0123456789ABCDEF";
```

443

```
    char *string = zmalloc (sizeof (uuid_t) * 2 + 1);
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        string [byte_nbr * 2 + 0] = hex_char [uuid [byte_nbr] >> 4];
        string [byte_nbr * 2 + 1] = hex_char [uuid [byte_nbr] & 15];
    }
    return string;
}
```

对等节点类的构造函数和析构函数如示例 8-9 所示。

示例8-9：UDP ping接口（interface.c）：对等节点的构造函数和析构函数

```
static peer_t *
peer_new (uuid_t uuid)
{
    peer_t *self = (peer_t *) zmalloc (sizeof (peer_t));
    memcpy (self->uuid, uuid, sizeof (uuid_t));
    self->uuid_str = s_uuid_str (self->uuid);
    return self;
}

//  销毁对等节点对象

static void
peer_destroy (peer_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        peer_t *self = *self_p;
        free (self->uuid_str);
        free (self);
        *self_p = NULL;
    }
}
```

示例 8-10 中的方法以二进制数据格式或可打印字符串形式返回对等节点的 UUID。

444 示例8-10：UDP ping接口（interface.c）：对等节点方法

```
static byte *
peer_uuid (peer_t *self)
{
    assert (self);
    return self->uuid;
}

static char *
```

```
peer_uuid_str (peer_t *self)
{
    assert (self);
    return self->uuid_str;
}

//  只是重新设置对等节点的过期时间，一旦我们
//  从一个对等节点获得任何活动就调用此方法

static void
peer_is_alive (peer_t *self)
{
    assert (self);
    self->expires_at = zclock_time () + PEER_EXPIRY;
}

//  一旦我们从代理节点删除对等节点，或删除该散列表
//  对等节点散列就自动调用本处理函数

static void
peer_freefn (void *argument)
{
    peer_t *peer = (peer_t *) argument;
    peer_destroy (&peer);
}
```

示例 8-11 中显示的结构为我们的代理保存上下文，所以我们可以干净地将其传递给需要它的方法。

示例8-11：UDP ping接口（interface.c）：代理类

```
typedef struct {
    zctx_t *ctx;                //  CZMQ 上下文
    void *pipe;                 //  返回应用程序的管道
    udp_t *udp;                 //  UDP 对象
    uuid_t uuid;                //  作为二进制大对象的 UUID
    zhash_t *peers;             //  已知节点的散列值，便于快速查找
} agent_t;
```

我们的代理的构造函数和析构函数如示例 8-12 所示。每个接口都有一个代理对象，它实现了后台线程。

示例8-12：UDP ping接口（interface.c）：代理的构造函数和析构函数 445

```
static agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
```

```
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->udp = udp_new (PING_PORT_NUMBER);
    self->peers = zhash_new ();
    uuid_generate (self->uuid);
    return self;
}

static void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        zhash_destroy (&self->peers);
        udp_destroy (&self->udp);
        free (self);
        *self_p = NULL;
    }
}
...
```

示例8-13显示了如何处理进入我们的UDP套接字的信标，这可能是来自其他对等节点的，或是我们自己的广播信标的回应。

示例8-13：UDP ping接口（interface.c）：处理信标

```
static int
agent_handle_beacon (agent_t *self)
{
    uuid_t uuid;
    ssize_t size = udp_recv (self->udp, uuid, sizeof (uuid_t));

    //  如果我们获得了一个 UUID 并且它不是我们自己的信标，我们就有一个对等节点
    if (size == sizeof (uuid_t)
    &&  memcmp (uuid, self->uuid, sizeof (uuid))) {
        char *uuid_str = s_uuid_str (uuid);

        //  通过其 UUID 字符串找到或创建对等节点
        peer_t *peer = (peer_t *) zhash_lookup (self->peers, uuid_str);
        if (peer == NULL) {
            peer = peer_new (uuid);
            zhash_insert (self->peers, uuid_str, peer);
            zhash_freefn (self->peers, uuid_str, peer_freefn);
```

```
            // 报告加入网络的对等节点
            zstr_sendm (self->pipe, "JOINED");
            zstr_send (self->pipe, uuid_str);
        }
        // 来自对等节点的任何活动表示它是活跃的
        peer_is_alive (peer);
        free (uuid_str);
    }
    return 0;
}
```
446

示例 8-14 所示的方法检查一个对等节点是否过期，如果现在对等节点还没有发送给我们任何东西，它就是“死”的，而我们可以将其删除。

示例8-14：UDP ping接口（interface.c）：获得对等节点

```
static int
agent_reap_peer (const char *key, void *item, void *argument)
{
    agent_t *self = (agent_t *) argument;
    peer_t *peer = (peer_t *) item;
    if (zclock_time () >= peer->expires_at) {
        // 报告对等节点离开网络
        zstr_sendm (self->pipe, "LEFT");
        zstr_send (self->pipe, peer_uuid_str (peer));
        zhash_delete (self->peers, peer_uuid_str (peer));
    }
    return 0;
}
```

后台代理的主循环如示例 8-15 所示。它采用 `zmq_poll()` 来监控前端管道（来自 API 的命令）和后端的 UDP 处理程序（信标）。

示例8-15：UDP ping接口（interface.c）：代理主循环

```
static void
interface_agent (void *args, zctx_t *ctx, void *pipe)
{
    // 创建用于传递的代理实例
    agent_t *self = agent_new (ctx, pipe);

    // 立即发送第一个信标
    uint64_t ping_at = zclock_time ();
    zmq_pollitem_t pollitems [] = {
        { self->pipe, 0, ZMQ_POLLIN, 0 },
        { 0, udp_handle (self->udp), ZMQ_POLLIN, 0 }
    };
```

```
    while (!zctx_interrupted) {
        long timeout = (long) (ping_at - zclock_time ());
        if (timeout < 0)
            timeout = 0;
        if (zmq_poll (pollitems, 2, timeout * ZMQ_POLL_MSEC) == -1)
            break;              //  中断

        //  如果我们在管道上有活动，则转去处理控制消息。
        //  现有的代码永远不会发送控制消息。
        if (pollitems [0].revents & ZMQ_POLLIN)
            agent_control_message (self);

        //  如果我们在 UDP 套接字上有输入，则转去处理该输入
        if (pollitems [1].revents & ZMQ_POLLIN)
            agent_handle_beacon (self);

        //  如果经过了 1 秒的标志时间，则广播我们的信标
        if (zclock_time () >= ping_at) {
            udp_send (self->udp, self->uuid, sizeof (uuid_t));
            ping_at = zclock_time () + PING_INTERVAL;
        }
        //  删除并报告任何过期的对等节点
        zhash_foreach (self->peers, agent_reap_peer, self);
    }
    agent_destroy (&self);
}
```

447

当我在两个窗口中运行这个程序时，它会报告一个节点正在加入网络。如果清除那个节点，几秒后它就告诉我该节点已经离开：

```
--------------------------------------
[006] JOINED
[032] 418E98D4B7184844B7D5E0EE5691084C
--------------------------------------
[004] LEFT
[032] 418E98D4B7184844B7D5E0EE5691084C
```

基于 ØMQ 消息的 API 的好处是，我可以把这个用我喜欢的任何方式包装。举例来说，如果我真的想做这些，可以把它变成回调方法。我还可以很容易地跟踪API上的所有活动。

有关调整有一些注意事项。在以太网中，5 秒(我在这个代码中用到的到期时间)似乎很长。而在一个压力很大的 WiFi 网络中，你可能会得到 30 秒以上的 ping 延迟。如果使用过于激进的到期值，你会断开仍然存在的节点。另一方面，最终用户应用程序期望一定的活跃度。如果需要花 30 秒来报告一个节点已经离开，那么用户将变得恼火。

一个体面的策略是迅速检测并报告失踪的节点，但只有在一个较长的时间间隔之后才将其删除。在视觉上，一个节点在它活着时将是绿色的，然后在它触不到时就灰一会儿，后来终于消失了。现在我们不这样做，但我们会在正在做的尚未命名的框架的实际实现中做到这一点。

如同将在后面看到的，我们必须把来自节点的任何输入，而不仅仅是 UDP 信标，作为它活着的一个标志。当存在很多 TCP 流量时，UDP 可能会受到挤压。这也许是我们没有使用现有的 UDP 发现库的主要原因：我们必须将这与我们的 ØMQ 消息传递紧密结合 448
才能让它正常工作。

关于 UDP 的更多内容

因此，我们有了在 UDP 的 IPv4 广播上工作的发现和存在。虽然这还不是最理想的，但它对于我们今天拥有的本地网络是有效的。但是，如果我们不添加额外的工作使其可靠，就不能在实际工作中使用 UDP。存在一个关于 UDP 的笑话，但有时你会碰到它，有时候却不会。

我们将坚持对所有一对一的消息使用 TCP。在发现后面还有一个 UDP 用例，这是多播文件分发。我会解释原因和方法，然后将其搁置到另一天来做。原因很简单：就是我们所说的“社交网络”只是增强的文化。我们通过共享创造文化，而这意味着越来越多的共享工作，我们制作出照片、文件、合同、消息，或对其进行组合。我们的设备云的目标向着做更多的这种工作迈进，而不是更少。

现在，分享的内容有两种主要模式。一种是发布 - 订阅模式，其中，一个节点对一组其他节点发出内容，在同一时间对所有节点都发送。另一种是后期加入者模式，其中一个节点稍微迟一点到达，而想要跟上谈话。我们可以使用 TCP 单播处理后期加入者，但在同一时间对一组客户端执行 TCP 单播有一些缺点。首先，它可能比多播慢。其次，它是不公平的，因为有些人会在别人前面得到内容。

在你进一步设计一个 UDP 多播协议前，必须认识到这不是一个简单的计算。当你发送多播数据包时，WiFi 接入点采用了低位速率，以确保即使是最远的设备也将安全地得到它。大多数正常的 AP 不做下面这个明显的优化，即测量最远设备的距离并使用该位速率。相反，它们只是用一个固定值。所以，如果你有接近 AP 的少量设备，多播将会出奇的慢。但如果你有一屋子的设备，它们都想得到这本书的下一章，多播就会出奇的高效。

该曲线相交于大约 6 ~ 12 台设备，这取决于网络。可以在理论上测量实时的曲线，并创建一个自适应协议。这将会很酷，但可能太难了，即使是对于最聪明的我们。

如果你坐下来勾画一个 UDP 多播协议，就必须意识到你需要一个用于恢复的通道，用

来获得丢失的数据包。你可能会想通过 TCP，使用 ØMQ 来做这件事。但就目前而言，我们会忘记多播 UDP，并假设所有流量都通过 TCP 来传输。

分拆一个库项目

在这个阶段，代码量增长到了比一个例子应有的规模更大，所以是时候来建立一个适当的 GitHub 项目了。这是一个规则：在公众视野中构建项目，并告诉人们你对项目所做
449 的进展，使得你的营销和社区建设工作从第一天就开始。我将会遍历涉及的所有内容。在第 6 章中我解释过培育围绕项目的社区的有关内容。我们需要下面几样东西：

- 一个名字
- 一个口号
- 一个公共 GitHub 存储库
- 一个链接到 C4 过程的自述文件
- 许可文件
- 一个问题跟踪器
- 两个维护者
- 第一个启动程序版本

首先是名字和口号。21 世纪的商标是域名，所以，在分拆一个项目时我做的第一件事就是寻找可能工作的一个域名。相当随机，我们的旧移动项目之一被称为“Zyre”，而我拥有它的域名。

我对于太张扬地将新项目推入 ØMQ 社区有点害羞，而且通常我会在我的个人账户或 iMatix 组织中启动一个项目。但是，我们已经了解到，在项目变得流行后移动它们起到的是反效果。我对未来充满了运动部件的预测要么是有效的，要么是错误的。如果这一章是有效的，我们不妨从一开始就将它作为 ØMQ 项目推出。如果它是错误的，我们以后可以删除该存储库，或者让它沉到准备遗忘的一个长长的清单的底部。

让我们从基础开始。协议（UDP 和 ØMQ/TCP）名称将是 ZRE（ZeroMQ 实时交换协议），而项目名将是 Zyre。我需要第二个维护人员，所以我邀请我的朋友 Dong Min（用纯 Java 编写一个名为 JeroMQ 的 ØMQ 栈的韩国黑客）加入。他一直工作在非常类似的想法上，所以也是如此热心。我们讨论这个问题，我们得到了同时在 JeroMQ 之上及 CZMQ 和 `libzmq` 之上构建 Zyre 的想法。这将使得 Zyre 更容易在 Android 上运行。它也会从一开始就给我们两个完全不同的实现，这对一个协议总是一件好事。

因此，采用我们在第 7 章构建的 FileMQ 项目，并把它作为一个新的 GitHub 项目的模板。GNU 的 autoconf 工具相当不错，但它有一个令人痛苦的语法。复制现有的项目文件，

并修改它们，这是最简单的。FileMQ 项目建立了一个库，并具有测试工具、许可文件、手册页，等等。这个项目不太大，所以这是一个很好的起点。

我在一个自述文件中总结了该项目的目标，并指向 C4。默认情况下，在新的 GitHub 项
目启用议题跟踪器，所以一旦把 UDP *ping* 代码作为第一个版本推出了，我们就已经准 450
备就绪了。然而，招募更多的维护者，这是一件好事，所以我创建了一个议题，"征求
维护者"，上面写着：

> 如果你想帮助一下那个可爱的绿色的"合并提取请求"按钮，并获得永恒的回报，请添加注释以确认你已经阅读并理解了位于 *http://rfc.zeromq.org/spec:16* 的 C4 过程。

最后，我改变了议题跟踪标签。默认情况下，GitHub 提供了各种通常的议题类型，但 C4 不使用它们。相反，我们只需要两个标签（红色的"紧急"和黑色的"就绪"）。

点对点消息传递

我们将利用我们最后的 UDP *ping* 程序并在它上面建立一个点对点消息传递层。我们的目标是能够在节点加入和离开网络时检测它们，传送消息给它们，并获得应答。这个问题解决起来是不简单的，这需要花费 Min 和我两天时间，才能得到一个能正常工作的"Hello World"版本。

我们必须解决一些问题：

- 在 UDP 信标中要发送什么样的信息，以及如何将它格式化。
- 使用什么类型的 ØMQ 套接字来连接节点。
- 发送什么 ØMQ 消息，以及如何设置它们的格式。
- 如何将消息发送到特定的节点。
- 如何知道任何消息的发送者，使得我们可以发送应答。
- 如何从丢失的 UDP 信标恢复。
- 如何避免信标使网络超负荷。

我将足够详细地解释这些，以便你明白我们过去所做的每一个选择都是为什么，并用一些代码片段来说明。我们将这个代码标记为 0.1.0 版本（*https://github.com/zeromq/zyre/zipball/v0.1.0*），所以你可以查看代码：困难的工作大部分是在 *zre_interface.c* 中完成的。

UDP 信标帧

在网络上发送 UUID 是最起码的逻辑寻址方案。但是，把这应用于实际工作之前，我们

还需要做其他几个方面的工作：

- 我们需要一些协议识别机制，使我们可以检查并拒绝无效的数据包。
- 我们需要一些版本信息，以便我们可以随着时间的推移更改这个协议。
- 451 我们需要告诉其他节点如何通过 TCP 联系我们，即它们可以与我们在上面交流的 ØMQ 端口。

让我们从信标消息格式开始着手。我们可能需要一个永远不会在未来版本中改变的固定的协议标头，以及一个取决于版本的正文（参见图 8-1）。

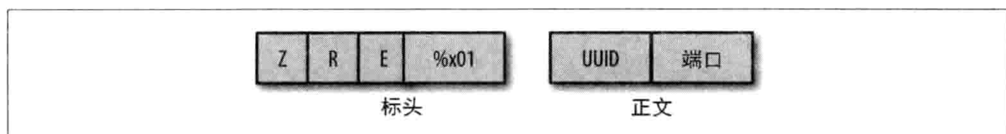


图8-1：ZRE发现消息

版本可以是 1 字节的从 1 开始的计数器。UUID 是 16 字节，而端口是一个 2 字节的端口号，由于 UDP 很好地将我们收到的每个消息的发送者的 IP 地址告诉了我们，这使我们得到了一个 22 字节的帧。

C 语言（和其他几种语言，像 Erlang）可以很方便地读取和写入二进制结构。我们将信标帧结构定义如下：

```
#define BEACON_PROTOCOL     "ZRE"
#define BEACON_VERSION      0x01

typedef struct {
    byte protocol [3];
    byte version;
    uuid_t uuid;
    uint16_t port;
} beacon_t;
```

这使得发送和接收信标都相当简单。下面是发送信标的方法，我们使用 zre_udp 类执行不可移植的网络调用：

```
//  信标对象
beacon_t beacon;

//  格式化信标字段
beacon.protocol [0] = 'Z';
beacon.protocol [1] = 'R';
```

```
beacon.protocol [2] = 'E';
beacon.version = BEACON_VERSION;
memcpy (beacon.uuid, self->uuid, sizeof (uuid_t));
beacon.port = htons (self->port);

//  将信标广播给正在监听的任何节点
zre_udp_send (self->udp, (byte *) &beacon, sizeof (beacon_t));
```

当接收信标时，我们需要警惕虚假数据。我们不会偏执地抵御攻击，如拒绝服务攻击。只是希望确保当一个坏的 ZRE 实现给我们发送错误帧时，我们不会崩溃。 452

为了验证一个帧，我们检查它的大小和标头。如果这些都 OK，我们就假设正文是可用的。当得到一个不属于我们自己的 UUID 时（回忆一下，我们会取回我们自己发的 UDP 广播），我们就可以把这个作为一个对等节点：

```
//  从网络获取信标帧
beacon_t beacon;
ssize_t size = zre_udp_recv (self->udp, (byte *) &beacon, sizeof (beacon_t));

//  在帧上执行基本验证
if (size != sizeof (beacon_t)
||  beacon.protocol [0] != 'Z'
||  beacon.protocol [1] != 'R'
||  beacon.protocol [2] != 'E'
||  beacon.version != BEACON_VERSION)
    return 0;               //  忽略无效信标

//  如果得到了一个 UUID，且这不是我们自己的信标，我们就有一个对等节点
if (memcmp (beacon.uuid, self->uuid, sizeof (uuid_t))) {
    char *identity = s_uuid_str (beacon.uuid);
    s_require_peer (self, identity,
        zre_udp_from (self->udp), ntohs (beacon.port));
    free (identity);
}
```

真正的对等连接（和谐模式）

由于 ØMQ 旨在简化分布式消息传递，人们经常会询问如何互联一组真实的节点（相对于明显的客户端和服务器）。这是一个棘手的问题，而 ØMQ 并没有真正提供一个明确的答案。

TCP 是 ØMQ 中最常用的传输协议，它是不对称的，一方必须绑定而另一方必须连接，虽然 ØMQ 试图对此保持中立，但它不是。当你连接时，将创建一个传出消息管道。当你绑定时，不创建管道。当没有管道时，你就不能写消息（ØMQ 将返回 EAGAIN）。

研究 ØMQ 的开发者然后会尝试建立对等节点集合之间的 N 对 N 的连接，它们常常试图用 ROUTER 到 ROUTER 流。原因是显而易见的：每一个节点需要寻址一系列的节点，这就需要 ROUTER。这通常以发到列表的哀怨的电子邮件结束。

经验告诉我们，ROUTER 到 ROUTER 特别难以成功使用。至少，一个对等节点必须绑定而另一个必须连接，这意味着该架构是不对称的。但同时，你根本不能指出何时你被允许安全地将消息发送给对等节点。这是一个第 22 条军规(译者注：意为不可逾越的障碍，两难处境)：你可以在一个节点跟你说话后跟它说话，但那个节点不能跟你说话，除非
453 你已经对它说话。一方或另一方就会失去信息，因而必须重试，这意味着这些对等节点不可能相等。

我将要解释和谐模式，它解决了这个问题，并且我们在 Zyre 中使用的就是它。

我们要保证的是，当对等节点“出现”在我们的网络中时，我们可以安全地与它交流，而 ØMQ 不会丢弃消息。对于这一点，我们必须使用一个连接到该对等节点的 DEALER 或 PUSH 套接字，这样即使该连接需要一些时间，那里也有立即可用的管道，而 ØMQ 将接受发出的消息。

DEALER 套接字不能分别满足多个对等节点。但是，如果我们对每一个节点都有一个 DEALER，并且我们将该 DEALER 连接到对等节点，那么一旦连接到对等节点，就可以放心地发送消息给它。

那么，下一个问题就是要知道谁给我们发了一个特定的消息。我们需要一个应答地址，这是发送任何给定消息的节点的 UUID。除非对每一个消息加上那个 16 字节的 UUID 的前缀，否则 DEALER 无法做到这一点，而这将是很浪费的。如果我们连接到路由器之前设置正确的身份，那么 ROUTER 套接字可以做到这一点。

这样一来，和谐模式归结为：

- 一个 ROUTER 套接字。我们将它绑定到一个临时端口，这是在我们的信标中广播的。
- 每个对等节点一个 DEALER 套接字，我们把它连接到对等节点的 ROUTER 套接字。
- 从我们的 ROUTER 套接字读取。
- 写入对等节点的 DEALER 套接字。

接下来的问题是，发现不是整齐地同步的。我们可能会在开始从一个对等节点接收消息后从它获得第一个信标。一个消息来自于 ROUTER 套接字，并具有连接到它上面的一个很好的 UUID，但没有物理 IP 地址和端口。我们必须在 TCP 上面强迫发现。要做到这一点，发给我们连接到的任何新的节点的第一个命令，是一个带有我们的 IP 地址和端口的 OHAI 命令。这确保了接收器在试图向我们发送任何命令前连接回我们。

把这分解成各个步骤：

- 如果收到一个 UDP 信标，我们就连接到该对等节点。
- 从我们的 ROUTER 套接字读取消息，每个消息都带有发送者的 UUID。
- 如果它是一个 OHAI 消息，我们连接回到那个节点（如果我们还没有连接到它）。
- 如果它是任何其他消息，我们就必须已经连接到该节点（这是一个放置断言的好地方）。
- 我们使用专用的节点 - 节点 DEALER 套接字给每个节点发送消息，它必须是已连接 454 的。
- 当连接到一个节点时，我们也告诉我们的应用程序该对等节点是存在的。
- 每次从一个节点得到一个消息时，我们都可以把它看成是一个信号检测（它还活着）。

如果我们不使用 UDP，而使用某种其他的发现机制，我宁愿还是对一个真正的对等网络用和谐模式：一个 ROUTER 用于从所有节点输入，并且每个节点一个 DEALER 用于输出。绑定 ROUTER，连接 DEALER，并用提供返回的 IP 地址和端口的 OHAI 等价物开始每个对话。我们需要一些外部机制来启动每个连接。

检测失踪

信号检测听起来很简单，但事实并非如此。当有大量的 TCP 流量时，UDP 数据包就会被丢弃，所以如果我们依赖于 UDP 的信标，就会得到虚假的断开。如果网络真的很繁忙，TCP 流量可延迟 5 秒、10 秒，甚至 30 秒。因此，如果我们当节点变得安静时清除它们，将会有虚假的断开。

因为 UDP 信标是不可靠的，人们很容易去添加 TCP 信标。毕竟，TCP 将可靠地传送它们。然而，这里有一个小问题。想象一下，你有一个包含 100 个节点的网络，并且每个节点每秒一次地发送一个 TCP 信标。每个信标都是 22 字节，不包括 TCP 的组帧开销。这是每秒 100*99*22 字节，或 217,000 字节 / 秒，而这只是信号检测。这大约是典型无线网络理想容量的 1% ~ 2%，这听起来不错。但是，当网络压力大时，或者是与其他网络争夺空域，额外的每秒 200K 将破坏其余的工作内容。UDP 广播至少是低成本的。

所以，我们做的是，只有当一个特定的节点一段时间后还没有发给我们任何 UDP 信标时才切换到 TCP 信号检测。然后，我们只对一个对等节点发送 TCP 信号检测。如果对方继续沉默，我们可以得出它已消失的结论。如果对等节点回来的时候带有一个不同的 IP 地址和 / 或端口，必须断开我们的 DEALER 套接字并重新连接到新的端口。

这给了我们每个对等节点的一组状态，但在这个阶段的代码还没有使用正式的状态机：

- 对等节点由于 UDP 信标而可见（我们使用来自信标的 IP 地址和端口连接）。

- 对等节点由于 OHAI 命令而可见（我们使用来自命令的 IP 地址和端口连接）。
- 对等节点似乎还活着（我们最近通过 TCP 得到一个 UDP 信标或命令）。
- 对等节点似乎是沉默的（在一段时间内没有活动，所以我们发送一个 HUGZ 命令）。
- 455 对等节点已经消失了（我们的 HUGZ 命令没有应答，所以我们销毁对等节点）。

在这个阶段的代码中我们还有一种剩余情况没有处理。对等节点改变 IP 地址和端口，而无须实际触发失踪事件，这是可能的。例如，如果用户关闭 WiFi 并随后切换回打开状态，则接入点可能会给节点分配一个新的 IP 地址。需要通过对 ROUTER 套接字解除绑定并在我们可以的时候重新绑定，来处理节点上消失了的 WiFi 接口。由于现在这不是设计的重点，我决定在 GitHub 追踪器上记录一个议题，并保留它，以备不时之需。

群发消息

群发消息是一种常见且非常有用的模式。这个概念很简单：你不是与单个节点交流，而是跟一“群”节点交流。该群只是一个名字，是你在应用程序中达成一致的一个字符串。这正如在 PUB 和 SUB 套接字中使用发布 - 订阅前缀。其实，我说“群发消息”，而不是“发布 - 订阅”的唯一原因是为了防止混淆，因为我们不打算在这里使用 PUB-SUB 套接字。

PUB-SUB 套接字几乎能够工作了。但是，我们刚刚做了这么多工作来解决后期加入者问题。在将消息发送到组之前，应用程序不可避免地会等待对等节点到达，所以我们要在和谐模式上构建，而不是再次在它的旁边启动。

来看看我们想要对群做的操作：

- 我们想加入和离开群。
- 我们想知道在任何给定的群中的其他节点是什么。
- 我们希望将消息发送给一个群（其中的所有节点）。

对使用互联网中继聊天（IRC）的任何人，这些内容都很眼熟，除了我们没有服务器外。每个节点需要对各群代表什么进行跟踪。此信息在网络上不会永远完全一致，但它将足够接近。

我们的接口将跟踪一组群（每个群一个对象）。这些群都是已知的，含有一个或多个成员节点，但不包括我们自己。当节点离开和加入群时，我们将跟踪它们。由于节点可以随时加入网络，我们要告诉新节点我们有什么群。当一个对等节点消失时，我们就会从我们所知道的所有群中删除它。

这给了我们一些新的协议命令：

JOIN

当我们加入一个群时，我们把这个命令发送给所有节点。

LEAVE 456

当我们离开一个群时，我们把这个命令发送给所有节点。

另外，我们在发送的第一个命令中增加一个 `groups` 字段（从现在开始，将 OHAI 更名为 HELLO，因为我需要一个更大的命令动词词汇）。

最后，让我们为对等节点添加一个方法来仔细检查它们的群数据的准确性。风险在于，我们会错过上述消息之一。虽然我们使用了和谐模式来避免在启动时典型的消息丢失，但偏执是值得的。现在，我们需要的是一种检测这种故障的方法。如果问题确实发生了，那么我们会在后面恢复。

我们将对这个使用 UDP 信标。我们要的是一个滚动计数器，它只是简单地告诉我们一个节点出现了多少的加入和离开操作（“转换”）。它从 0 开始，并对每个我们加入或离开的组递增。我们可以用一个最小的 1 字节的值，因为这会捕获所有的失败，除了像天文数字般罕见的失败“我们正好连续失去了 256 个消息”（译者注：一个字节能表示 0~255 的值）（这是在第一个演示过程中碰到的）。我们还将把转换计数器放入 JOIN、LEAVE 和 HELLO 命令，并试图发起这个问题，我们将通过加入 / 离开几百个群来测试，使用设置为 10 左右的高水位标记。

现在到为群发消息选择动词的时间了。我们需要一个意思是“跟一个节点说”的命令，还需要一个意思是“跟很多节点说”的命令。经过一番尝试，我最好的选择是 WHISPER 和 SHOUT，这就是程序代码使用的。SHOUT 命令需要告诉用户群名，以及发送者节点。

由于群类似于发布 - 订阅，你可能也想使用这个来广播 JOIN 和 LEAVE 命令，也许是通过创建一个所有节点都加入的“全局性”群。我的建议是把群纯粹作为用户空间的概念来保持，有两个原因。首先，如果你需要全局群发出一个 JOIN 命令，你怎么加入全局群呢？其次，它在特殊情况下（保留的名称）会造成混乱。

明确地发送 JOIN 和 LEAVE 到所有连接的节点，这是更简单的。

我不打算详细介绍群发消息的实现，因为它是相当迂腐和不精彩的。对群和对等节点进行管理的数据结构不是最佳的，但它们是可行的。我们需要：

- 接口的群列表，我们可以在一个 HELLO 命令中将它发送到新的节点。
- 其他节点的群的散列，这是我们用来自 HELLO、JOIN 和 LEAVE 命令的信息更新的。
- 每个群的对等节点的散列，我们用相同的三个命令来更新它。

457 在这个阶段，我开始得到相当令人满意的二进制序列化（我们从第 7 章得到的编解码器生成器），它处理列表和字典，以及字符串和整数。

这个版本在仓库中被标记为 v0.2.0，并且如果你想检查代码在这个阶段看起来是什么样的，可以下载压缩包（*https://github.com/zeromq/zyre/tags*）。

测试与模拟

当你用部件来建立一个产品，并且这包括一个类似 Zyre 的分布式框架时，要了解它是否会在现实生活中正常工作的唯一方式是在每一个部件上都模拟实际活动。

使用断言

正确使用断言是成为一名专业的程序员的标志之一。

我们作为创建者的偏见使得正确地测试我们的工作很困难。我们倾向于编写测试来证明代码能够正常工作，而不是试图证明事实并非如此。这有很多原因。我们对自己和别人来假装我们可以是（可能是）完美的，而事实上，我们一贯犯错误。代码中的错误被视为“坏”，而不是“必然”，所以在心理上，我们希望更少地看到它们，而不是更多地发现它们。“他编写了完美的代码”被看作是一种恭维，而不是“他从来不冒险，所以他的代码像冷面条一样乏味和被大量使用”的一种委婉的说法。

在教育和工作中，有些文化教我们向往完美和惩罚错误，这使得这种态度更差。要接受我们都会犯错误这个现实，然后学习如何把它转换成利润，而不是耻辱，这在任何行业中都是最难的智力训练之一。我们通过与他人合作并且通过更快，而不是更晚地挑战我们自己的工作，充分利用我们的错误。

一个可以简化这项工作的技巧是使用断言。断言不是错误处理的一种形式，它们是可执行的实际理论。代码断言，“在这一时刻，如此这般必须为真”，如果断言失败，代码会杀死自己。

你能越快地证明代码不正确，你就可以越快越准确地解决它。相信代码正常工作，并证明其行为符合预期是不太科学且更不可思议的思维。能够说“`libzmq` 有 500 个断言，并且虽然我尽了最大努力，但它们无一失败。”比这好很多。

因此，在 Zyre 代码库中散布了断言，特别是处理节点状态的代码中的两个断言。这是最难做对的方面：节点需要跟踪对方，并准确交换状态，否则程序就会停止工作。该算法依赖于到处乱飞的异步消息，而且我敢肯定最初的设计有缺陷。

而当我通过手工启动和停止 `zre_ping` 实例测试原 Zyre 代码的时候，每隔一段时间我就会得到一个断言失败。手工运行往往不能足够频繁地重现这些失败，所以让我们来做一个适当的测试工具。 458

前期测试

如果能够在实验室中对各个组件的实际行为进行充分测试，就可以让你的项目费用产生 10 倍或 100 倍的差异。工程师对自己工作的偏见使得前期测试非常有利可图，而后期测试的代价高得令人难以置信。

我将告诉你一个小故事，它是关于我们在 20 世纪 90 年代末曾工作过的一个项目的。在那个项目中，我们为一个工厂自动化项目提供软件，而其他团队提供硬件。三四支团队把他们的专家带到现场，这是在偏远地区的一家工厂（污染工厂怎么总是在一个偏远的边境）。

其中一个团队，是一家专门从事工业自动化的公司，负责建造自动售票机、售票亭，以及在它们上面运行的软件。没有什么不寻常的：刷证件，选择一个选项，收到票据。他们在现场组装了两个售票亭，每星期带来一些更多的零碎东西。票据打印机、显示器屏幕、来自以色列的特殊键盘。这些东西必须是防灰尘的，因为售票亭设在外面。但是没有一样东西能正常工作。在阳光下看不见屏幕上显示的内容。票据打印机不断卡住并打印出错误的东西。亭子的内部设施只是放在木制的架子上。售票亭软件经常崩溃。这是喜剧，但该项目真的，*真的*必须要工作，所以我们花了几个星期，甚至几个月，现场帮助其他团队调试他们的零碎东西，直到它正常工作。

一年后，第二家工厂，同样的故事又上演了。这时候，客户已经等得不耐烦了。因此，又一年后，当到了第三家也是最大的工厂时，我们跳了起来，说："请把售票亭和软件以及一切东西都交给我们来做。"

我们对软件和硬件进行了详细设计，并为所有部件找到了供应商。我们花了三个月的时间在互联网上搜索每个组件，并且再用两个月，将它们组装成每块重约 20 公斤的不锈钢砖。这些砖是 60 厘米见方的并有 20 厘米高，在牢不可破的玻璃后面有一块巨大的平面面板，还有两个接口：一个用于电源，一个用于以太网。请你将纸槽中装上足够六个月用的打印纸，然后把钢砖拧紧在壳体上，它会自动开机，发现其 DNS 服务器，并加载它的 Linux 操作系统，然后是应用软件。它连接到实际的服务器并显示主菜单。你可以通过刷特殊的证件，然后输入代码获得对配置屏幕的访问。

它的软件是可移植的，所以我们可以在编写它时测试，而且我们收集了来自供应商的部件，我们对每个部件都保留一个，所以我们有一个拆散的亭可以使用。当我们得到了我

459 们的成品亭时，它们都能立刻开始工作。我们发货给客户，他们将它们插入到壳体中，启动它们，并开始处理业务。我们在现场花了一个星期左右，并在 10 年中，只有一个售票亭损坏（屏幕坏掉后被替换）。

得到的教训是，我们要在前期执行测试，这样一来，当你插入某个东西时，你就会知道它将会如何表现。如果你还没有在前期测试它，那么你会在现场花费几个星期或几个月，爆出本来就不应该有的问题。

Zyre 测试仪

在我做过的手动测试中，很少会触发一个断言。然后，它消失了。因为我不相信魔法，这意味着该代码某个地方仍然是错的。因此，下一步是对 Zyre v0.2.0 代码的高强度测试，试图破坏它的断言，并弄清它在现场会怎么表现。

我们将发现和消息传递功能作为一个接口对象打包，它被主程序创建、使用，然后销毁的。我们不使用任何全局变量，这使得可以很容易地启动大量的接口来模拟真实的活动，所有这些都在一个进程中执行。如果我们已经从编写大量的例子中学会了一件事情，那就是 ØMQ 在单个进程中协调多个线程的能力比多个进程使用起来要容易得多。

测试仪的第一个版本包括启动和停止一组子线程的一个主线程，每个子线程运行一个接口，每一个接口带有一个 ROUTER、DEALER 和 UDP 套接字（参见图 8-2 中的 R、D 和 U）。

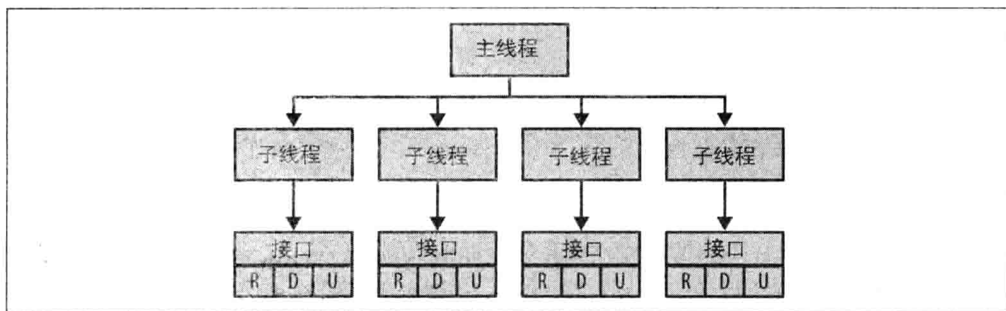


图8-2：Zyre测试工具

测试仪的好处是，当我连接到一个 WiFi 接入点，所有 Zyre 流量（即使是同一进程中的两个接口之间的）都会通过 AP。这意味着我可以完全只用在一个房间里的两台 PC 对任何无线网络基础设施执行压力测试。怎么强调它的可贵都不为过：比如说，如果把 Zyre 建立为 Android 的一个专用服务，我们最终就需要几十台 Android 平板电脑或手机进行任何大规模的测试。售票亭，诸如此类。

现在的重点是破坏目前的代码，试图证明它是有错误的。在这个阶段中测试的重点不是它运行得有多好，它的速度有多快，使用多少内存，或其他任何东西。我们将逐步尝试破坏各个功能（并遭到失败），但首先我们将尝试破坏一些已经在代码中投入的核心断言。 460

它们是：

- 任何节点接收的来自对等节点的第一个命令必须是 HELLO。换句话说，对等网络连接过程不能丢失消息。
- 每个节点为其对等节点计算的状态都与每个对等节点为自己计算的状态相匹配。换句话说，再次重复，在网络中没有丢失消息。
- 当应用程序将消息发送到对等节点时，我们拥有了一个到该对等节点的连接。换句话说，只有我们已经建立了一个到对等节点的 ØMQ 连接后，应用程序才能“看到”这个对等节点。

在使用 ØMQ 时，有几种我们可能会丢失消息的情况。第一种情况是发生“后期加入者”综合征。第二种情况是我们关闭套接字而没有发送任何东西。第三种情况是我们溢出了 ROUTER 或 PUB 套接字的高水位标记。而第四种情况是我们将一个未知的地址与 ROUTER 套接字联用。

现在，我认为和谐模式解决了所有这些可能的情况。但是，我们还要往这个组合中增加 UDP。因此，测试仪的第一个版本模拟不稳定的和动态的网络，其中节点随机地来来去去。正是在这里，事情会破坏。

下面是测试仪的主线程，它负责管理一个包含 100 个线程的池，随机启动和停止每一个线程。每隔大约 750 毫秒，它要么启动，要么停止一个随机的线程。我们随机化这个时间，使线程不是全都同步的。几分钟后，平均 50 个线程愉快地彼此聊着，就像韩国的青少年在江南地铁站那样：

```
int main (int argc, char *argv [])
{
    //  初始化交流任务的上下文
    zctx_t *ctx = zctx_new ();
    zctx_set_linger (ctx, 100);

    //  获取要模拟的接口数量（默认值为 100）
    int max_interface = 100;
    int nbr_interfaces = 0;
    if (argc > 1)
        max_interface = atoi (argv [1]);

    //  我们把接口处理为管道的数组
```

```
    void **pipes = zmalloc (sizeof (void *) * max_interface);

    //  我们将随机启动和停止接口线程
    while (!zctx_interrupted) {
461     uint index = randof (max_interface);
        //  切换接口线程
        if (pipes [index]) {
            zstr_send (pipes [index], "STOP");
            zsocket_destroy (ctx, pipes [index]);
            pipes [index] = NULL;
            zclock_log ("I: Stopped interface (%d running)", --nbr_interfaces );
        }
        else {
            pipes [index] = zthread_fork (ctx, interface_task, NULL);
            zclock_log ("I: Started interface (%d running)", ++nbr_interfaces );
        }
        //  随机休眠大约 750 毫秒，所以我们使活动平滑
        zclock_sleep (randof (500) + 500);
    }
    zctx_destroy (&ctx);
    return 0;
}
```

需要注意的是，我们对每个子线程都要维护一个管道（当我们使用 zthread_fork() 方法时，CZMQ 自动创建管道）。正是通过这条管道，当是时候让子线程离开时，我们告诉它们停止。子线程执行下列操作（为了清晰起见，我切换到伪代码）：

```
create an interface
while true:
    poll on pipe to parent, and on interface
    if parent sent us a message:
        break
    if interface sent us a message:
        if message is ENTER:
            send a WHISPER to the new peer
        if message is EXIT:
            send a WHISPER to the departed peer
        if message is WHISPER:
            send back a WHISPER 1/2 of the time
        if message is SHOUT:
            send back a WHISPER 1/3 of the time
            send back a SHOUT 1/3 of the time
    once per second:
        join or leave one of 10 random groups
destroy interface
```

这段伪代码的含义如下：

```
创建一个接口
当条件为真：
    在到父母的管道上和接口上轮询
    如果父母给我们发消息：
        跳出
    如果接口给我们发消息：
        如果消息是 ENTER：
            发送 WHISPER 到新的对等节点
        如果消息是 EXIT：
            发送 WHISPER 给离开的对等节点
        如果消息是 WHISPER：
            在 1/2 的时间发回 WHISPER
        如果消息是 SHOUT：
            在 1/3 的时间发回 WHISPER
            在 1/3 的时间发回 SHOUT
    每秒一次：
        加入或退出 10 个随机组之一
销毁接口
```

测试结果

是的，我们破坏了代码。实际上，破坏了好几次。这是令人满意的。我会努力解决我们找到的不同问题。

使得节点就一致的群状态达成一致是最困难的。每个节点都需要跟踪整个网络的群成员，正如我在“群发消息”一节已经解释的。群发消息是一种发布 - 订阅模式。JOIN 和 LEAVE 类似于订阅和退订消息。这对这些曾经丢失的节点是至关重要的，否则我们会发现节点随机从群中离开。 462

因此，每个节点计算它做过的 JOIN 和 LEAVE 的总数，并且把这种状态（如 1 个字节的滚动计数器）在其 UDP 的信标中广播。其他节点拿起这个状态，并将其与自己计算的值进行比较，如果这两者有区别，代码就引发断言。

第一个问题是，UDP 信标会随机延迟，所以它们对于承载状态是无用的。当一个信标迟到时，状态是不准确的，我们得到了一个漏报。为了解决这个问题，我们将状态信息移到 JOIN 和 LEAVE 命令中，也把它添加到 HELLO 命令中。该逻辑就变成：

- 从一个节点的 HELLO 命令获取其初始状态。
- 当从一个节点获得 JOIN 或 LEAVE 时，递增状态计数器。

- 检查新状态计数器的值是否与在 JOIN 或 LEAVE 命令中的值相匹配。
- 如果不匹配，则引发断言。

我们遇到的下一个问题是，消息在新的连接上意外到达。和谐模式首先连接，然后发送 HELLO 作为第一个命令。这意味着，接收方应该总是从一个新的节点得到一个 HELLO 作为第一个命令。相反，我们看到 PING、JOIN 和其他命令首先到达。

事实证明，这是由于 CZMQ 的临时端口的逻辑造成的。临时端口仅仅是一个服务可以得到的一个动态分配的端口，而不用要求一个固定的端口号。一个 POSIX 系统通常在 0xC000 到 0xFFFF 的范围分配临时端口。CZMQ 的逻辑是在这个范围内寻找一个空闲端口，绑定到该端口，并把端口号返回给调用者。

这听起来不错，直到你密集地停止一个节点并启动另一个节点，新节点得到旧节点的端口号的时候。请记住，ØMQ 试图重新建立已断开的连接。因此，当第一个节点停止时，它的对等节点将重试连接它。当新的节点出现在同一个端口上时，顿时所有的节点都连接到它，并开始聊天就像他们是老朋友。

这是影响任何较大规模的动态 ØMQ 应用程序的普遍问题。它有许多可行的解决方案。一是不要重复使用临时端口，当你在一个系统上有多个进程时，这说起来容易做起来难。另一种解决办法是每次都选择一个随机端口，这至少减少创建刚刚释放的端口的风险。这带来了一个可能性也许下降到 1/1000 的垃圾连接风险，但它仍然存在。也许最好的解决办法是接受这可能发生的现实，了解其原因，并在应用程序级别处理它。

463 我们有一个有状态的协议，它总是从一个 HELLO 命令开始。我们知道，节点连接到我们，以为我们是离去又回来的一个现有节点，并发送给我们其他命令，这是可能的。第一步是，当我们发现一个新的对等节点时，就销毁连接到相同端点的任何现有对等节点。这不是一个圆满的答案，但至少它是有礼貌的。第二步是，无视从一个新的对等节点进来的任何东西，直到该节点说 HELLO。

这不需要对协议进行任何修改，但必须在协议中规定当我们遇到它时怎么做：由于 ØMQ 连接的工作方式，它可能会从行为良好的节点收到意想不到的命令，而没有方法返回一个错误代码，或以其他方式告诉该节点重置其连接。因此，对等节点必须抛弃来自对等节点的任何命令，直到它收到一个 HELLO。

事实上，如果你把这画在一张纸上，并仔细考虑，你会看到，你永远不会从这样的连接得到一个 HELLO。该节点会发送 PING、JOIN 和 LEAVE，并最终超时和关闭，因为它没有从我们这里得到任何返回的检测信号。

你还可以看到，有产生混乱的风险，在我们的 DEALER 套接字上，没有办法把从两个对

等节点来的命令组合成单个流。

当你满意这个作品时，我们准备继续前进。这个版本在存储库中被标记为 v0.3.0，如果你想检查在这个阶段代码看起来什么样，你可以下载压缩包。

需要注意的是，做大量节点的高负荷模拟可能会导致你的程序用尽文件句柄，从而在 libzmq 中产生一个断言失败。我通过（在我的 Linux 电脑）运行下面的命令将每个进程的限制提高到 30000：

```
ulimit -n 30000
```

跟踪活动

要调试在这里看到的这种问题，我们需要广泛的日志记录。有很多是并行发生的，但每一个问题都可以追踪到两个节点之间的具体交流，这是由一组按严格的顺序发生的事件组成的。我们知道如何做出非常复杂的记录，但像往常一样，只做真正需要的，而不是更多的记录，才是明智的。我们必须捕获：

- 每个事件的时间和日期。
- 事件在哪个节点上发生。
- 对等节点，如果有的话。
- 事件是什么（例如，到达的是哪个命令）。
- 事件数据，如果有的话。

最简单的方法是打印所需的信息到控制台，并带上时间戳。这是我使用的方法。然后， 464
就可以很容易地找到受故障影响的节点，从日志文件中筛选出只指向它们的消息，看看到底发生了什么。

处理阻塞节点

在任何对性能敏感的 ØMQ 架构中，你都需要解决流量控制问题。不能简单地发送无限量信息到一个套接字并希望得到最好的结果。在一个极端，你可能会耗尽内存。这是一个消息代理的经典故障模式：一个缓慢的客户端停止接收消息，代理开始对消息排队，最终耗尽内存并使整个进程死掉。在另一个极端，套接字删除消息，或阻塞，因为你触及了高水位标记。

我们希望使用 Zyre 把消息分发到一组节点，并且要公平地做到这一点。使用一个单独的 ROUTER 接口来输出将是有问题的，因为任何一个被阻塞的对等节点都会阻止给所有的对等节点输出流量。TCP 确实具有良好的算法来在一组连接中分布网络容量。我们正

在使用一个单独的 DEALER 套接字去跟每个对等节点交流，所以从理论上说，每一个 DEALER 套接字都将在后台公平合理地发送队列中的消息。

一个 DEALER 套接字触及它的高水位标记的正常行为是阻塞。这通常是理想的，但我们的问题就在这里。目前的接口设计使用一个线程，该线程将消息分发给所有的节点。如果其中的一个发送调用是阻塞的，那么所有输出都将阻塞。

有几个选项可用来避免阻塞。一种是在一整套 DEALER 套接字上使用 zmq_poll()，而且只写入准备就绪的套接字。我不喜欢这样，有两个原因。首先，DEALER 套接字隐藏在对等节点类里面，而让每一个类都来明确地处理这个会更整洁。其次，对我们还不能提供给 DEALER 套接字的消息，我们做什么处理呢？我们在哪里对它们排队呢？第三，它似乎是绕过了此问题。如果对等节点是真的那么繁忙而不能读它的消息，那么一定有什么东西出错了。最有可能的是，它已经死掉了。

所以，对输出没有轮询。第二个选择是对每个对等节点使用一个线程。我很喜欢这个想法，因为它适合“在一个线程中做一件事”的 ØMQ 设计模式，但这将在模拟中创建很多的线程（我们启动的节点数的平方），而且我们已经用尽了文件句柄。

第三种选择是使用一个非阻塞发送。这更好，而且它是我选择的解决方案。然后，我们可以为每个对等节点提供大小合理的传出队列（该 HWM），并且如果已满，就把它作为对等节点的致命错误。这将适用于较小的消息。如果我们要发送大块，例如，对于内容分发，需要在上面用一个基于信用的流量控制机制。

465 因此，第一步是要证明给自己，我们可以把正常阻塞的 DEALER 套接字转变成一个非阻塞套接字。示例 8-16 创建一个正常的 DEALER 套接字，它连接到某个端点（所以没有传出管道而套接字将接受消息），将高水位标记设置为 4，然后将发送超时时间设置为 0。

示例8-16：对DEALER套接字检查EAGAIN（eagain.c）

```
//
//  显示如何在达到 HWM 时引发 EAGAIN
//
#include <czmq.h>

int main (void) {
    zctx_t *ctx = zctx_new ();

    void *mailbox = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_set_sndhwm (mailbox, 4);
    zsocket_set_sndtimeo (mailbox, 0);
    zsocket_connect (mailbox, "tcp://localhost:9876");
```

```
    int count;
    for (count = 0; count < 10; count++) {
        printf ("Sending message %d\n", count);
        int rc = zstr_sendf (mailbox, "message %d", count);
        if (rc == -1) {
            printf ("%s\n", strerror (errno));
            break;
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

当我们运行这个程序时，我们成功地发送 4 条消息（它们无处可去，套接字只是将它们排队），然后得到一个不错的 EAGAIN 错误：

```
发送消息 0
发送消息 1
发送消息 2
发送消息 3
发送消息 4
资源暂时不可用
```

下一个步骤是为对等节点确定一个合理的高水位标记。Zyre 表示人类的交互，即，以低频率聊天的应用程序，例如两个游戏或共享的绘图程序。我期望每秒 100 条消息是相当多的。我们的“对等节点是真的死了”超时时间为 10 秒，所以，1000 的高水位标记似乎是合适的。

不是设定一个固定的 HWM，或使用默认值（它随机也可能恰好是 1000），我们将它作为 100* 超时计算。下面是我们如何为一个对等节点配置新的 DEALER 套接字的办法： 466

```
// 创建新的输出套接字（删除在传输途中的任何消息）
self->mailbox = zsocket_new (self->ctx, ZMQ_DEALER);

// 设置我们的调用者“From”身份，使接收节点知道
// 每个消息都是谁传来的
zsocket_set_identity (self->mailbox, reply_to);

// 设置一个允许合理活动的高水位标记
zsocket_set_sndhwm (self->mailbox, PEER_EXPIRED * 100);

// 立即发送消息或返回 EAGAIN
zsocket_set_sndtimeo (self->mailbox, 0);

// 连接到对等节点
```

```
zsocket_connect (self->mailbox, "tcp://%s", endpoint);
```

最后，当我们在对等节点上得到一个 EAGAIN 时，应该做些什么呢？我们不需要去执行销毁对等节点的一切工作，因为接口会自动这样做，如果它在超时时间内没有从对等节点得到任何消息。但只是去掉最后一条消息似乎很脆弱：它会给接收节点一些缺口。

我宁愿得到一个更粗暴的回应。粗暴是好事，因为它迫使做出一个“好”或“坏”的决定，而不是一个模糊的“应该工作，但说实话，它有很多的边界情况让我们担心以后。”销毁套接字，断开对等节点，并停止发送任何东西给它。对等节点最终将不得不重新连接并重新初始化任何状态。这是一种“每秒 100 条消息对任何人都足够”的断言。所以，在 zre_peer_send() 方法中，我们这样做：

```
int
zre_peer_send (zre_peer_t *self, zre_msg_t **msg_p)
{
    assert (self);
    if (self->connected) {
        if (zre_msg_send (msg_p, self->mailbox) && errno == EAGAIN) {
            zre_peer_disconnect (self);
            return -1;
        }
    }
    return 0;
}
```

其中 disconnect 方法看起来像这样：

```
void
zre_peer_disconnect (zre_peer_t *self)
{
    // 如果连接，销毁套接字并丢弃所有未决消息
467 assert (self);
    if (self->connected) {
        zsocket_destroy (self->ctx, self->mailbox);
        free (self->endpoint);
        self->endpoint = NULL;
        self->connected = false;
    }
}
```

分布式日志记录和监视

让我们来看看日志记录和监视。如果你曾经管理一个真正的服务器（如 Web 服务器），

你就知道拥有捕获正在发生什么的能力是多么重要。有一个长长的原因清单，特别是如下几点：

- 为了测量随时间推移的系统性能。
- 要了解什么样的工作做得最多，以优化性能。
- 要跟踪误差和它出现的频度。
- 要对故障做事后检查。
- 要在发生争端时提供审计线索。

从我们认为必须要解决的问题角度来对此划定范围：

- 我们要跟踪重要事件（如节点离开和重新加入网络）。
- 对于每个事件，我们要跟踪一组一致的数据：日期/时间，观察到该事件的节点，产生该事件的对等节点，事件本身的类型，以及其他事件数据。
- 我们希望能够随时切换登录和注销。
- 我们希望能够以机械方式处理日志数据，因为这将是相当大量的。
- 我们希望能够监视正在运行的系统，也就是说，实时收集日志，并对它们进行分析。
- 我们希望将记录流量的工作对网络的影响降到最小。
- 我们希望能够在网络上的一个点收集日志数据。

与任何设计一样，其中的一些要求是相互冲突的。例如，实时收集日志数据意味着它在网络上发送，这会在一定程度上影响网络流量。然而，在任何设计中，这些要求也都是假设，除非我们有运行的代码，所以我们不能把它们太当回事儿。我们将目标定在代码貌似足够好，并随着时间的推移而改善上。

一个合理的最小实现 468

可以说，只将日志数据转储到磁盘是一个解决方案，它是大多数移动应用程序所做的（使用“调试日志”）。但大多数故障需要来自两个节点的相关事件。这意味着手工搜索大量的调试日志，以找到那些重要的事件。这不是一个很聪明的做法。

我们希望立即或适时地（如存储和转发）将日志数据发送到某个集中的地方。现在，让我们专注于即时记录日志。我的第一个想法是，当涉及发送数据时，就用 Zyre 来完成这个任务：只把日志数据发送到一个名为“LOG”的组，并希望有人收集它。

但使用 Zyre 去记录 Zyre 的日志本身就是一件两难的事。谁记录这个记录器的日志？如果我们希望发送每个消息的详细日志会怎么样？我们是否在它里面包含记录日志消息呢？这很快就会变得混乱。我们希望有一个独立于 Zyre 的主 ZRE 协议的日志记录协议。最简单的方法是使用发布-订阅协议，其中所有节点都在 PUB 套接字上发布日志数据，

而收集器通过一个 SUB 套接字挑选那个数据（参见图 8-3）。

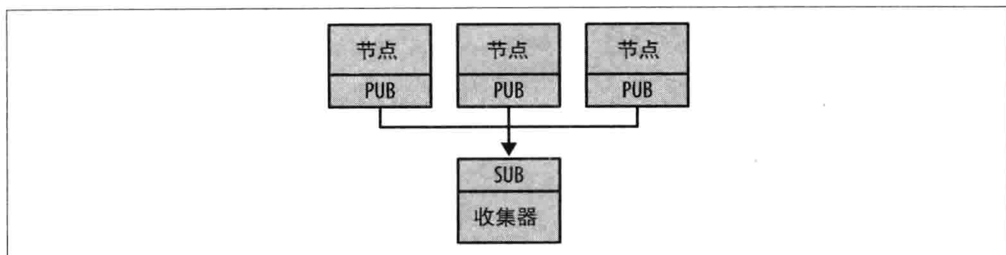


图8-3：分布式日志收集

当然，该收集器可以运行在任何节点上。这给了我们的用例一个很好的范围：

- 被动日志收集器，它在磁盘上存储日志数据，用于最终的统计分析。这会是一台硬盘空间足够保存数周或几个月日志数据的 PC。
- 一个收集器，它将日志数据存储到可以被其他应用程序实时使用的一个数据库中。这对小型工作组可能是大材小用，但对于跟踪较大的组，它的表现是非常出色的。收集器可以通过 WiFi 收集日志数据，然后通过以太网将它转发到某个地方的数据库中。
- 一个现场测量应用程序，它加入 Zyre 网络，然后实时从节点收集日志数据，显示事件和统计信息。

接下来的问题是如何互联节点和收集器。哪一方绑定，哪一方连接？这两种方法在这里都会工作，但如果 PUB 套接字连接到 SUB 插口，那么稍好一些。如果你还记得，ØMQ
469 的内部缓冲区仅当有连接时才会突然形成。这意味着，只要一个节点连接到收集器，它就可以开始无丢失地发送日志数据。

我们如何告诉节点连接到哪些端点呢？在网络上，我们可以有任意数量的收集器，它们将使用任意的网络地址和端口。我们需要某种服务公告机制，在这里我们可以使用 Zyre 为我们做这项工作。可以使用群发消息，但把服务发现构建到 ZRE 协议本身似乎更整洁。这没什么复杂的：如果一个节点提供服务 X，在它发送给其他节点一个 HELLO 命令时，它可以告诉它们这些。

我们将用保存一组“名称 = 值”对的 *headers* 字段来扩展 HELLO 命令。让我们来定义标头 `X-ZRELOG` 来指定收集器端点（SUB 套接字）。例如，一个充当收集器的节点可以添加这样的标头：

```
X-ZRELOG=tcp://192.168.1.122:9992
```

当另一个节点看到这个标头时，它只是将它的 PUB 套接字连接到那个端点。日志数据现在被分发给网络上的所有收集器（零个或多个）。

做这第一个版本是非常简单的，只用了半天时间。下面是我们不得不做出或更改的部分：

- 我们制作了一个新的类，zre_log，接受日志数据并管理到收集器的连接，如果有的话。
- 我们对节点标头增加了一些基本的管理，取自 HELLO 命令。
- 当一个节点具有 X-ZRELOG 标头时，我们就连接到它指定的端点。
- 当记录日志到标准输出时，我们通过 zre_log 类切换到日志记录。
- 我们用一个让应用程序设置标头的方法扩展接口 API。
- 我们编写一个简单的管理 SUB 套接字，并设置 X-ZRELOG 标头的记录器应用程序。
- 当发送 HELLO 命令时，我们发送自己的标头。

这个版本在 Zyre 存储库中被标记为 V0.4.0，如果你想检查在这个阶段代码看起来的样子，可以下载压缩包。

此刻的日志消息仅仅是一个字符串。我们过一会儿将制作更专业的结构化的日志数据。

首先，动态端口的说明。在用来测试的 *zre_tester* 应用程序中，我们积极地创建并销毁接口。一个后果是，一个新的接口可以很容易地重复使用刚刚由另一个应用程序释放的端口。如果某个地方有一个 ØMQ 套接字试图连接到这个端口，其结果可能是热闹的。

下面是我曾遇到的情景，这引起了几分钟的混乱。该记录器在一个动态的端口上运行： 470

1. 启动记录器应用程序。
2. 启动测试器应用程序。
3. 停止记录器。
4. 测试器接收无效的消息（并如设计那样地断言）。

当测试器创建了一个新接口时，该接口重复使用（刚刚停止）记录器释放来的动态端口，并且突然接口开始从它的邮箱节点接收日志数据。我们之前看到过类似的情况，在那里一个新接口可以重用一个旧接口释放的端口，并开始获取旧接口的数据。

这个教训是，如果你使用动态端口，那么请准备从重新连接到你的消息不灵通的应用程序接收随机数据。切换到静态端口虽然停止了行为不端的连接，但这不是一个圆满的解决方案。它有如下两个缺点：

- 当我写这本书时，libzmq 在连接时不检查套接字的类型。ZMTP/2.0（*http://rfc.zeromg.com/spec:15*）协议不公布每个节点的套接字类型，所以这个检查是可行的。
- ZRE 协议没有快速失败（断言）机制，我们需要先读取并解析整个消息才能识别出

它是否无效。

让我们来解决第二个问题。套接字对的校验无论如何都不能完全解决这个问题。

协议断言

正如维基百科（*http://en.wikipedia.org/wiki/Fail_Fast*）所说的那样，"快速失败的系统通常被设计为停止正常运行，而不是试图继续一个可能有缺陷的过程"。一个类似 HTTP 的协议具有快速失败的机制，其中客户端发送到一个 HTTP 服务器的前 4 个字节必须是"HTTP"。如果不是这样，服务器可以关闭连接而不读取任何东西。

我们的 ROUTER 套接字不是面向连接的，所以当我们得到不好的传入消息时没有办法"关闭连接"。不过，如果它不是有效的，我们能够扔掉整个消息。当使用临时端口时，这个问题将是更糟糕的，但它广泛地适用于所有协议。

所以，让我们把协议断言定义为我们在每一个消息开头放置的一个独特的签名，它标识预期的协议。当读到一条消息时，我们检查它的签名，如果签名不是我们所期望的，就默默地丢弃该消息。一个好的签名应该不与常规的数据混淆，并给了我们足够的空间用于表示多种协议。

471 我将使用一个 16 位的签名，包括一个 12 位的模式和 4 位的协议 ID（参见图 8-4）。模式 %xAAA 是为了区别于其他时候可能会在消息开头看到的值：%x00、%xFF 和可打印的字符。

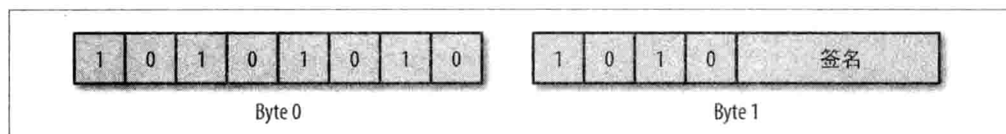


图8-4：协议签名

在生成协议的编解码器时，添加这一断言是相对容易的。其中的逻辑是：

- 获取消息的第一帧。
- 检查前两个字节是否是 %xAAA 和预期的 4 位签名的形式（译者注：2 字节用 2*8=16 个二进制位表示，%xAAA 占用了 3 个十六进制数字，即 12 个二进制位，还剩 4 个二进制位用于签名）。
- 如果是这样，那么继续解析消息的其余部分。
- 如果不是，跳过所有"更多"帧，获取第一帧，然后重复。

为了测试，我切换回使用临时端口的记录器。该接口现在能够正确检测并丢弃缺少一个有效签名的任何消息。如果消息有一个有效签名，但仍然是错误的，那么这是一个真正的 bug。

二进制日志记录协议

现在，我们使日志框架正常工作了，让我们来看看协议本身。在网络中发送字符串很简单，但是当涉及 WiFi 时，我们真的不能再浪费带宽。我们有高效的使用二进制协议的工具，所以让我们设计一个用于记录日志的工具。

这将是一个发布 - 订阅协议，并且在 ØMQ v3.x 中，我们做发行方过滤。这意味着如果把日志记录级别放在消息的开头，那么就可以做多级日志（错误、警告、信息）。因此，我们的消息开始于一个协议签名（2 字节）、一个记录日志级别（1 字节）和一个事件类型（1 字节）。

在第一个版本中，我们发送 UUID 字符串来标识每个节点。作为文本，这些 UUID 每个都是 32 个字符长。可以发送二进制的 UUID，但它仍然是冗长和浪费的。在日志文件中我们不关心节点标识符，需要的只是用来关联事件的一些方法。那么什么是我们可以使用的对于记录日志足够唯一的最短标识符呢？我说“足够唯一”，是因为虽然我们真的想在实际的代码中使 UUID 重复的可能性为零，但日志文件并没有那么严格。

最简单可行的答案是将 IP 地址和端口散列转换为一个 2 字节的值。我们会遇到一些冲突，但它们将是罕见的。这有多么罕见呢？作为一种快速完整性的检查，我写了一个小程序，
它产生了一堆地址并将它们散列成 16 位的值，以寻找冲突。为了确信这一点，我产生 472
跨少量 IP 地址（匹配模拟设置）的 10000 个地址，然后是跨大量 IP 的地址（匹配现实生活中的设置）。此程序用的散列算法是一种修改后的伯恩斯坦算法：

```
uint16_t hash = 0;
while (*endpoint)
    hash = 33 * hash ^ *endpoint++;
```

多次运行后我没有遇到任何冲突，所以这将作为日志数据的一个标识符。这增加了 4 个字节（两个字节被节点用于记录该事件，而另两个字节在来自对等节点的事件中用于它的对等节点）。

接下来，我们要存储事件的日期和时间。POSIX `time_t` 类型曾经是 32 位的，但因为这将在 2038 年溢出，它现在是一个 64 位的值。我们会利用这一点，因为没有必要在一个日志文件中使用毫秒分辨率：事件是连续的，时钟是不太可能紧密同步的，而且网络延迟意味着精确的时间都不是那么有意义的。

我们最多用 16 字节，这是体面的。最后，我们允许一些额外的数据，它们被格式化为文本，并取决于事件的类型。综上所述提供了以下消息规范：

```
<class
    name = "zre_log_msg"
    script = "codec_c.gsl"
    signature = "2"
>
这是 ZRE 日志记录协议 - 原始版本。
<include filename = "license.xml" />

<!-- 协议常量 -->
<define name = "VERSION" value = "1" />

<define name = "LEVEL_ERROR" value = "1" />
<define name = "LEVEL_WARNING" value = "2" />
<define name = "LEVEL_INFO" value = "3" />

<define name = "EVENT_JOIN" value = "1" />
<define name = "EVENT_LEAVE" value = "2" />
<define name = "EVENT_ENTER" value = "3" />
<define name = "EVENT_EXIT" value = "4" />

<message name = "LOG"id = "1">
    <field name = "level" type = "number" size = "1" />
    <field name = "event" type = "number" size = "1" />
    <field name = "node" type = "number" size = "2" />
    <field name = "peer" type = "number" size = "2" />
    <field name = "time" type = "number" size = "8" />
    <field name = "data" type = "string" />
Log an event
</message>

473 </class>
```

它为我们产生 800 行完美的二进制编解码器（zre_log_msg 类）。该编解码器执行协议断言，就像主 ZRE 协议一样。代码生成的入门曲线相当陡峭，但它极易于把你的设计从“业余”推向“专业”。

内容分发

现在，我们有了一个健壮的框架，它用于创建节点群、让它们互相聊天，并监控所产生的网络。接下来的步骤是让它们把内容分发为文件。

像往常一样，我们将瞄准非常简单可行的解决方案，然后再逐步提高。我们希望它至少包括以下内容：

- 应用程序可以告诉 Zyre API “发布这个”，并提供位于文件系统某处的文件的路径。
- Zyre 将这个文件发给所有的节点——包括那些那个时候在网络上的节点和那些以后到达的节点。
- 接口每次接收到文件时，它就告诉它的应用程序，“这里是这个文件”。

我们可能最终会希望有更多的辨别力，例如，发布到特定的群，但假如它真的需要，我们可以在以后补充那些功能。

在第 7 章中，我们开发了一种设计用于插入 ØMQ 应用程序的文件分发系统（FileMQ）。让我们来使用它。

每个节点都将是一个文件发布者，以及文件订阅者。我们将发布者绑定到一个临时端口（如果使用 FileMQ 的标准端口 5670，就不能在一台电脑上运行多个接口），并且广播在 HELLO 消息中的发布者的端点，就像我们对日志收集器所做的。这可以让我们互联所有节点，使所有订阅者跟所有发布者交流。

需要确保每个节点都有自己的目录，用于发送和接收文件（发件箱和收件箱）。再次重复，这是我们可以在一台电脑上运行多个节点的方式。既然已经对每个节点都有了一个唯一的 ID，我们就只在目录名中使用它们。

下面是创建一个新的接口时，我们设置 FileMQ API 的方法：

```
sprintf (self->fmq_outbox, ".outbox/%s", self->identity);
mkdir (self->fmq_outbox, 0775);

sprintf (self->fmq_inbox, ".inbox/%s", self->identity);
mkdir (self->fmq_inbox, 0775);

self->fmq_server = fmq_server_new ();
self->fmq_service = fmq_server_bind (self->fmq_server, "tcp://*:*");
fmq_server_publish (self->fmq_server, self->fmq_outbox, "/");
fmq_server_set_anonymous (self->fmq_server, true);
char publisher [32];
sprintf (publisher, "tcp://%s:%d", self->host, self->fmq_service);
zhash_update (self->headers, "X-FILEMQ", strdup (publisher));

//  当客户端发现新节点时，它就连接
self->fmq_client = fmq_client_new ();
fmq_client_set_inbox (self->fmq_client, self->fmq_inbox);
```

474

```
fmq_client_set_resync (self->fmq_client, true);
fmq_client_subscribe (self->fmq_client, "/");
```

并且当我们处理一个 HELLO 命令时，检查 X-FILEMQ 标头字段：

```
//  如果对等节点是一个 FileMQ 发布者，则连接到它
char *publisher = zre_msg_headers_string (msg, "X-FILEMQ", NULL);
if (publisher)
    fmq_client_connect (self->fmq_client, publisher);
```

最后一件事是在 Zyre API 中公开内容分发。我们需要两样东西：

- 一种让应用程序说“发布这个文件”的方法。
- 一种让接口告诉应用程序“我们接收到了这个文件”的方法。

从理论上说，应用程序可以仅仅通过在发件箱目录中创建符号链接来发布文件，但我们使用一个隐藏的发件箱，这有点困难。所以我们添加一个 API 方法，publish：

```
//  发布文件到虚拟空间
void
zre_interface_publish (zre_interface_t *self, char *pathname, char *virtual)
{
    zstr_sendm (self->pipe, "PUBLISH");
    zstr_sendm (self->pipe, pathname);
    zstr_send  (self->pipe, virtual);
}
```

API 将该方法传递到接口线程，这在发件箱目录中创建文件，以便 FileMQ 服务器将它捡起来，并广播它。我们可以最终将文件数据复制到这个目录，但由于 FileMQ 支持符号链接，所以我们用它来代替。该文件有一个“.ln”扩展名并包含一行，即实际路径名。

最后，我们怎么通知收件人文件已经到达了呢？ FileMQ fmq_client API 有一个这方面的消息（DELIVER），所以我们在 zre_interface 中必须做的就是从 fmq_client API 抓住这个消息，并把它传递给我们自己的 API：

```
zmsg_t *msg = fmq_client_recv (fmq_client_handle (self->fmq_client));
zmsg_send (&msg, self->pipe);
```

475 这段代码是复杂的，它一次做了很多工作。但我们只用了大约 10000 行代码就把 FileMQ 和 Zyre 组合在一起。最复杂的 Zyre 类，zre_interface，是 800 行代码，这是紧凑的。如果你仔细地正确组织它们，基于消息的应用程序都保持它们的形状。

编写 Unprotocol

我们已经拥有了一个正式的协议规范的所有部件，现在该是把协议写在纸上的时候了。这么做有两个原因：第一，以确保任何其他实现能正确地相互通话，二是因为我想为 UDP 发现协议获得一个官方端口，这意味着做文书工作。

像所有我们在本书开发的其他 unprotocol 一样，协议驻留在 ØMQ RFC 网站（*http://rfc:zeromy.org/spee:20*）。协议规范的核心是命令和字段的 ABNF 语法：

```
zre-protocol    = greeting *traffic

greeting        = S:HELLO
traffic        = S:WHISPER
                / S:SHOUT
                / S:JOIN
                / S:LEAVE
                / S:PING R:PING-OK

;   问候一个节点，以便它可以连回我们
S:HELLO         = header %x01 ipaddress mailbox groups status headers
header          = signature sequence
signature       = %xAA %xA1
sequence        = 2OCTET             ; 递增序列号
ipaddress       = string             ; 发送者的 IP 地址
string          = size *VCHAR
size            = OCTET
mailbox         = 2OCTET             ; 发送者邮箱端口号
groups          = strings            ; 列出发送者所在的群
strings         = size *string
status          = OCTET              ; 发送者群状态序列号
headers         = dictionary         ; 发送者标头属性
dictionary      = size *key-value
key-value       = string             ; 格式化为名称 = 值

; 发送消息给一个对等节点
S:WHISPER       = header %x02 content
content         = FRAME              ; 作为 0MQ 帧的消息内容

; 发送消息给一个群
S:SHOUT         = header %x03 group content
group           = string             ; 群名称
content         = FRAME              ; 作为 ØMQ 帧的消息内容

; 加入一个群
```

476

```
S:JOIN          = header %x04 group status
status          = OCTET           ; 发送者群状态序列号

; 离开一个群
S:LEAVE         = header %x05 group status

; ping 一个已经默默地离开的对等节点
S:PING          = header %06

; 应答一个对等节点的 ping
R:PING-OK       = header %07
```

结论

构建用于不稳定的\分散的网络的应用程序是 ØMQ 的残局之一。随着每一年计算的成本下降，这样的网络（无论是电子计算机或云中的虚拟机）变得越来越普遍。在这一章中，我们已经整合了本书介绍的许多技术来构建 Zyre，即通过本地网络进行邻近计算的框架。Zyre 不是唯一的，人们已经为应用程序打开此领域进行了很多尝试（ZeroConf、SLP、SSDP、UPnP、DDS）。但这些尝试似乎都因为过于复杂或很难被应用程序开发者用作构建的基础而告终。

Zyre 还没有完成。像许多在本书中的项目一样，它是其他项目的一个破冰船。还有一些主要领域尚未完成，下面是我们可以在本书或本软件以后的版本中解决的一些问题：

高级别 API

现在 Zyre 提供的基于消息的 API 是可以使用的，但仍然比我想为普通开发者提供的更复杂。如果有一个我们绝对不能错过的目标，那它就是原始的简单性。这意味着应该在许多语言中建立高级别的 API，它们隐藏所有的消息传递并将它们归结为简单的方法，如启动、加入/离开群、获取消息、发布文件和停止。

安全

如何才能建立一个完全分散式的安全系统呢？我们也许可以利用公钥基础设施来做一些工作，但这要求节点有自己的互联网接入，这是不能保证的。据我们所知，答案是使用任何现有的安全对等网络连接（TLS、蓝牙，也许是 NFC）来交换一个会话密钥，即一个对称加密口令。对称加密口令有其优点和缺点。

游牧内容

作为一个用户，我该如何管理跨多个设备的内容呢？ Zyre+FileMQ 组合可能有助于
477 本地网络中的使用，但我希望在互联网上也能够做到这一点。有没有我可以用的云服务呢？有某个来使用 ØMQ 的东西吗？

联盟

我们如何将一个局域分布式应用程序在全球范围内扩展呢？联盟是一个可能的答案，这意味着创建集群的集群。如果可以加入 100 个节点来共同创建一个本地集群，那么也许可以加入 100 个集群来共同创建一个广域集群。随后面临的挑战是那么相似：发现、存在、群发消息。

第 9 章 479

后记

番外篇

我请求这本书的一些贡献者告诉我们他们在用 ØMQ 做什么。以下是他们的故事。

Rob Gagnon 的故事

“我们使用 ØMQ 协助聚合每一分钟在我们的全球电信服务器的网络上发生的数千个事件，以使我们能够准确地报告和监测需要我们注意的情况。ØMQ 不仅方便了这套系统的开发，而且开发过程比我们在原始设计中计划的更快，系统也更健壮和容错。

“我们能够很容易地在网络中添加和删除用户，而没有丢失任何消息。如果我们需要提高系统的服务器部分，也可以停止并重新启动它，而不必担心要首先停止所有客户端。ØMQ 内置的缓冲功能使这一切都成为可能。”

Tom van Leeuwen 的故事

“我一直在寻找创建某种将各种服务连接在一起的服务总线的办法。已经有一些产品实现了一个代理，但它们并没有我所需要的功能。无意间，我偶然发现了 ØMQ，它真棒。它非常轻巧、精益、简单，且容易学习，因为这本书是非常完整的，读起来很不错。其实我已经实现了泰坦尼克模式和管家代理，还有一些补充（客户端 / 工人身份验证，以及通过发送目录来解释它们提供了什么以及它们应该如何被处理的工人）。

“ØMQ 的妙处事实上在于它是一个库，而不是一个应用程序。只要你喜欢，你可以塑造它，它只是把诸如排队、重新连接、TCP 套接字之类的无聊东西放在后台，确保你可以把精 480
力集中在对你重要的事情上。我用 Ruby 实现了各种类型的工人 / 客户端和代理，因为那

是我们用于开发的主要语言，我们还实现了一些 PHP 的客户端从现有的 PHP Web 应用连接到总线。我们为云服务使用这个服务总线，把各种平台的设备连接到服务总线的公开功能上以实现自动化。

“ØMQ 很容易理解，而如果你花了一天读这本书，你就会对它的工作原理有良好的认识。虽然我是一名网络工程师，而不是一个软件开发者，但我也成功地为我们的自动化需求完成了一个非常好的解决方案！ ØMQ：非常感谢你！”

Michael Jakl 的故事

“我们每天使用 ØMQ 在我们的分布式处理管道中分发数百万个文件。我们开始于大消息队列代理，但它们有自己各自的问题和困难。在简化我们的架构的追求中，我们选择了 ØMQ 来做接线。到目前为止，它在如何扩展架构，以及如何易于改变和移动组件方面产生了巨大的影响。它有大量的语言绑定，这使得我们能选择正确的工具来工作，而不会在我们的系统中牺牲互操作性。我们不使用大量的套接字（在我们的整个应用程序中小于 10），但是这已经完全满足了我们将一个巨大的整体式应用程序拆分成独立的小部件的需要。

“总而言之，ØMQ 让我自己保持清醒，并帮助我的客户把支出保持在预算范围之内。”

Vadim Shalts 的故事

“我是 ActForex 公司金融市场软件开发团队的领导者。由于工作领域的性质，我们需要快速处理大量的报价。此外，减少处理订单和报价的延迟时间是非常关键的，只实现高吞吐量是不够的。一切都必须在软实时与对每一个报价都可预测的超低延迟时间中处理。该系统由多个交换消息的组件组成。每个报价都可能经历很多处理阶段，每一个阶段都会增加总的延迟。因此，组件之间消息传递的可预测的低延迟成为我们的架构的一个关键因素。

“为了找到一个适合我们需要的解决方案，我们研究了不同的方案，尝试了不同的消息代理（RabbitMQ、ActiveMQ Apollo、Kafka），但其中任何一个都未能达到可预测的低延迟。最终，我们选择了 ØMQ 与 ZooKeeper 的结合来用于服务发现。使用 ØMQ 进行复杂的协调，需要一个比较大的工作量和对它的良好理解，这是多线程的自然复杂性的结果。我们发现，像 ZooKeeper 那样的外部代理是服务发现和协调的一个更好的选择，
481 而 ØMQ 可主要用于简单的消息传递。ØMQ 完美地融入了我们的架构，它允许我们用最少的努力来实现所需的延迟时间。它把我们从消息的处理瓶颈中拯救出来，并使得处理时间非常稳定且可预测。

“对于非常强调低延迟的解决方案，我可以断然推荐 ØMQ。”

本书是如何诞生的

当我开始写一本 ØMQ 书时，我们仍在 ØMQ 社区争论派生和提取请求的利弊。值得欣慰的是，今天，这种说法似乎固定了下来：我们在 2012 年年初采纳了 `libzmq` 的“自由主义”政策，打破了我们对单个首要作者的依赖，并为几十个新的贡献者开辟了空间。更深刻的，它让我们从旧的强制行军模式转移到一个迥然不同的渐进有机进化模型。

我之所以有信心这个方式可行，是因为我们在这份指南上一年或更长时间的工作已经展示了这种方式。诚然，文字组织是我自己的工作，这也许是因为它应该是。写作不是编程。当我们写作时，我们在讲一个故事，一个人不想用不同的声音讲一个故事，这感觉有点怪怪的。

对于我来说，这个项目的真正的长期价值是例子的存储库：用 24 种不同的语言编写的约 65,000 行代码，这促使更多的人访问 ØMQ。当人们想告诉别人如何学习 ØMQ 时，他们已经参考了 Python 和 PHP 示例库——这两个是最完整的库，但它也是对编程语言的学习。

例如，下面是在 Tcl 的代码中的循环：

```
while {1} {
    # 处理消息的所有部分
    zmq message message
    frontend recv_msg message
    set more [frontend getsockopt RCVMORE]
    backend send_msg message [expr {$more?"SNDMORE":""}]
    message close
    if {!$more} {
        break ; # 消息的最后部分
    }
}
```

以及在 Lua 中的相同的循环：

```
while true do
    -- 处理消息的所有部分
    local msg = frontend:recv()
    if (frontend:getopt(zmq.RCVMORE) == 1) then
        backend:send(msg, zmq.SNDMORE)
    else
        backend:send(msg, 0)
```

```
            break;         --  消息最后部分
482         end
        end
```

这个特殊的例子（*rrbroker*）也包含在这本书的在线版本中，用 C#、C++、CL、Clojure、Erlang、F#、Go、Haskell、Haxe、Java、Lua、Node.js、Perl、PHP、Python、Ruby、Scala、Tcl，当然还有 C 语言编写。这个代码库，全都在 MIT/X11 许可协议下开放源代码，它可以形成其他的书或项目的基础。

但是对于这个翻译的集合，最深刻的说法是这样的：你选择的语言是一个细节，甚至是一种分心。ØMQ 的力量在于它给你的和让你建立的模式，而这些模式超越语言的更迭。我作为一个软件和社会架构师的目标是建立可以持续几代人的结构。而瞄准仅仅几十年似乎是没有意义的。

消除摩擦

我将从消除摩擦的方面来解释我们使用的技术工具链。我们正通过这本书讲一个故事，而我们的目标是要尽可能廉价和顺利地让尽可能多的人得到它。

其核心思想是将这本书托管在 GitHub 上，使任何人都能轻松地做出贡献。然而，事实证明这比预计的更复杂。

让我们开始分工。我是一个优秀的作家，能迅速产生无穷数量像样的文字。但若要提供其他编程语言的例子，这是我力所不能及的。因为核心 ØMQ API 是用 C 语言编写的，用它来编写原始实例似乎合乎逻辑。同样，C 是一种中性的选择，它也许是唯一不使人产生强烈情感的语言。

如何鼓励人们将例子翻译成各种编程语言呢？我们尝试了一些方法，而最后工作得最好的方法是在正文的每一个例子上提供“选择语言”链接，这将人们带到该语言的翻译或转到一个解释他们如何能够为这个翻译做贡献的页面。它的通常工作方式是，当人们用他们的首选语言学习 ØMQ 时，他们也贡献了一些翻译，或修复现有的翻译。

与此同时，我注意到有少数人很坚定地翻译*每*一个例子。这些人主要是绑定作者，他们意识到例子是用来鼓励人们使用他们的绑定的一个伟大的方式。鉴于他们的努力，我扩展了脚本来制作这本书的特定语言的在线版本。在这些版本中，我们包括的不是 C 代码，而是 Python 或 PHP 代码。Lua 和 Haxe 也得到了它们的专门书籍。

一旦拥有谁在什么上面开展工作的想法，我们就会知道如何组织工作本身。很显然，对于编写和测试例子，你想用来工作的东西是*源代码*。所以在建立这本书的时候，我们导

入此源代码，而这就是我们如何制作特定语言版本书籍的方法。 483

我喜欢用纯文本格式来写作，它的速度快并与类似 Git 的源代码控制系统合作得很好。由于我们网站的主要平台是 Wikidot，因此我使用 Wikidot 的可读性非常好的标记格式来写作。

至少在前几章中，画图来解释消息在节点之间的流动是很重要的。我发现了 Ditaa，这是一种输入线条图并输出精美图案的可爱工具。在文本中具有图形，和文字一样，使得它非常容易使用。

至此，你会意识到，我们使用的工具链是高度定制的，虽然它使用了大量的外部工具。但所有的工具都可以在 Ubuntu 上获得，这是一种慈悲，整个工具链都在 *bin* 子目录的 *zguide* 存储库中。

接下来让我们介绍编辑和出版过程。我们产生本书在线版本的方法如下：

```
bin/mkguide
```

它的工作原理如下：

- 原始文本位于一系列的文本文件（每章一个）中。
- 例子位于 *examples* 子目录，根据每种语言分类。
- 我们取得这些文本并将它们加工成为一组对 Wikidot 准备就绪的文件，包括每种得到它们自己版本的语言。
- 我们提取图形并对每一个文件调用 Ditaa，以产生图像文件，这些文件存储在 *images* 子目录中。
- 我们提取内嵌清单（这是没有翻译的），并在 *listings* 子目录中存储这些。
- 我们对每个例子和清单使用 *pygmentize*，以产生 Wikidot 格式的标记页面。
- 我们使用 Wikidot API 上传所有更改后的文件到这本书的 wiki。

从头开始做这个需要一段时间。所以，我们存储每一个图像、清单、例子和文本文件的 SHA-1 签名，并只处理和上传变化的文件，并且这使得当人们做出新的贡献时，可以很容易地出版这本书的新版本。

为了生成 PDF 和 ePub 格式，我们的做法如下：

```
bin/mkpdfs
```

它的工作原理如下：

- 我们在所有输入文件上使用 *mkbook* 脚本产生 DocBook 输出。

484

- 在每一种语言中，我们通过 *docbook2ps* 和 *ps2pdf* 从 DocBook 格式创建整洁的 PDF 文件。
- 在每一种语言中，我们通过 *db2epub* 从 DocBook 格式创建 epub 电子书。
- 我们使用 Wikidot API 将 PDF 文件上传到 wiki。

当创建一个社区项目时，关键是要降低“更改的延迟”，这是指让人们看到他们的工作活跃，或者至少，看到你已经接受了他们的提取请求所需的时间。如果这个延迟超过一两天，你的贡献者往往就会失去对你的兴趣。

许可

我希望人们能在自己的本职工作中重新使用这个文本：在演讲、文章，甚至其他的书籍中。然而，这笔交易是，如果他们重新组合我的工作，那么其他人也可以重新组合他们的作品。我喜欢信用，并且不反对他人用他们的组合作品赚钱。因此，本书文本是根据 cc-by-sa（创作共用协议）授权的。

我们一开始对例子使用 GPL 许可，但我很快就意识到，这是不可行的。例子的重点是要给人们可重用的代码片段，以便他们更加广泛地使用 ØMQ，而如果这些都是 GPL 的，将不会产生这个效果。所以我们把例子的许可切换到 MIT/X11，即使对于那些理所应当可以采用 LGPL 许可的更大、更复杂的例子也是如此。

然而，当我们开始将例子转换成独立的项目（如管家）时，我们使用 LGPL。再次重复，可组合性胜过传播。许可证是工具，我们要利用它们的意图，而不是思想。

索引[†]

Symbols

0MQ (see ZeroMQ)

A

ABNF, 379
AMQP
  authentication for, 410
  messaging used by, 24
APIs
  contracts as, 374
  for discovery, 439–448
  for FileMQ project, 412, 416–417
  high-level, for ZeroMQ, 102–110
assertions, 56
  (see also testing)
  best practices for, 457
  protocol assertions, 470–471
asserts, removed by optimizer, 57
asynchronous client/server pattern, 111–116
asynchronous disconnected network, 194–206
Asynchronous Majordomo pattern, 186–191
authentication
  SASL for, 410–411
  state for, 405

B

Benevolent Tyrant role, 366
binary logging protocol, 471–473
Binary Star pattern, 206–222
  adding to Clone pattern, 296–306
  for reactor class, 218–222
  requirements for, 208–211
  split-brain syndrome, preventing, 211
binding (see server node)
Black Box pattern, 258–260
bridging, 54–56
broker (see proxy or broker)
brokerless messaging, reliability for, 223–243
BSD license, 330–332, 335
burnout, reducing risk of, 364–366

C

C string format, 10–11
C4 contract, 325, 335–349
Canary Watcher role, 368
Cheap or Nasty pattern, 380
CHP (Clustered Hashmap Protocol), 306–309
client node
  connecting sockets to endpoint, 32–34
  multiple, connecting to multiple servers with
    proxy, 143
    (see also Majordomo pattern; Paranoid
      Pirate pattern; Simple Pirate pattern;
      Titanic pattern)
  multiple, connecting to multiple servers
    without proxies, 144, 223–243

---

†：索引所列页码为本书英文版页码，请参照用“▭”表示的原书页码。

multiple, connecting to single server, 143, 144–148
role of, 32, 33, 89
starting before server node, 33
clocks, portable, 104
Clone pattern, 260–320
adding Binary Star pattern to, 296–306
central server for, 261
Clustered Hashmap Protocol for, 306–309
ephemeral values with, 284–292
multithreaded stack for, 310–320
reactor with, 292–295
requirements for, 260
state representation, 261–271
state snapshots, 271–276
state subtrees, 281–283
state updates from clients, 276–281
cloud computing example (see Inter-Broker Routing example)
Clustered Hashmap Protocol (CHP), 306
COD (Complexity-Oriented Design), 361–362
code examples, xv, 5
(see also patterns)
creation of, 481–484
Git repository for, 5
Hello World, 5–9
Inter-Broker Routing example (see Inter-Broker Routing example)
licensing for, 484
licensing of, xv, 5
Load-Balancing Message Broker, 96–102
Multiple Socket Reader/Poller, 41–44
Multithreaded Hello World, 65–67
Multithreaded Relay, 68–70
Parallel Task Ventilator, 16–20, 57–59
permission to use, xvi
Request-Reply Broker, 50–54
Synchronized Publisher, 71–74
translations of, xv, 5
Weather Update Proxy, 54–56
Weather Update Server, 11–15
website for, xv
Zyre project (see Zyre project)
code generation, 386–392
Collective Code Construction Contract (see C4 contract)
collector (see server node)
community, 325–327
building and maintaining, 332–335
burnout, reducing risk of, 364–366
C4 contract, 325, 335–349
example of, 349–352
iMatix's role in, 327
licensing, 330–332, 335, 339
open source model used by ZeroMQ, 325
software architecture guidelines, 327–335
structure of, 326–327
Complexity-Oriented Design (COD), 361–362
connectedness of software, 3–4
connecting (see client node)
connections, 32–34
(see also client node)
transports for, 33
Constant Gardner role, 367
contact information for this book, xvi
context
adding threads to, 36
best practices for, 21–22
configuring, 28
creating, 21
destroying, 21, 22, 57
monitoring, 28
as threadsafe, 64, 69
contracts, 163, 374
APIs as, 374
C4 contract, 325, 335–349
unprotocols as, 374, 375–382
contributors to this book, 479–481
conventions used in this book, xv
cooperative discovery, 434–438
Cost Gravity, 425
creation of this book, xiii–xv, 481–484
Ctrl-C (SIGINT), handling, 61–62, 105, 107, 110, 110
CZMQ API, 105–110

D

data serialization (see serialization of data)
DEALER and DEALER combination, 88, 114
DEALER and REP combination, 87
DEALER and ROUTER combination, 88
(see also ROUTER-DEALER proxy)
asynchronous client/server pattern using, 111–116
load balancing using, 94–102
DEALER socket, 38, 49, 86, 86–88
(see also ROUTER-DEALER proxy)

design models, 356–364
COD (Complexity-Oriented Design), 361–362
TOD (Trash-Oriented Design), 359–361
Digital Standards Organization (Digistan), 328
disconnected network, asynchronous, 194–206
disconnected TCP transport (see tcp transport)
discovery, 432–448
API for, 439–448
cooperative discovery, 434–438
preemptive discovery, 432–434
testing with multiple nodes per device, 439
UDP for, 448–448
distributed computing, 425–427
discovery, 432–448
API for, 439–448
cooperative discovery, 434–438
preemptive discovery, 432–434
testing with multiple nodes per device, 439
logging and monitoring for, 467–473
Zyre project for (see Zyre project)
dynamic discovery, 45–47

E

EAGAIN return code, 57, 466
Earth and Sky role, 366
EFSM error code, 87, 144
EHOSTUNREACH error code, 91
EINTR return code, 62
envelopes, 75–76, 81–86
ephemeral values, 284–292
error handling, 56–59
with Cheap and Nasty pattern, 382
by ROUTER socket, 91
Espresso pattern, 247–249
ETERM return code, 57
examples (see code examples)
exclusive pair pattern, 37
exiting, best practices for, 22–23

F

failure, causes of, 141
(see also reliability)
fair-queuing, 20
federation interconnection, 121
FileMQ project, 411–424
building, 414
client and server API for, 412, 416–417
configuration for, 418–419
delivery notifications for, 420
file stability, determining, 419
internal architecture of, 415–416
late joiners, handling, 421–422
maintaining state for, 417–418
protocol for, 413–414
recovery from failure for, 421–422
running, 415
symbolic links with, 420
testing, 423–424
used in Zyre project, 449, 473–475
FILEMQ protocol, 413–414
files
large-scale distribution of, 411–424, 473–475
transferring, 392–403
Flash Mob role, 368
flow control, 77
fonts used in this book, xv
Fork + Pull Model, GitHub, 335
fork() function, 21
Foundation for a Free Information Infrastructure (FFII), 328
frames, 40, 382
(see also multipart messages)
free software (see open source software)
Freelance pattern, 223–243

G

Gagnon, Rob (contributor), 479
Git branches, 352–355
GitHub, 338, 343
C4 contract with, 336
Fork + Pull Model, 335
issue tracker, 338, 343, 344
projects (see projects)
GPL license, 330–332, 335, 339, 378
group messaging pattern, 455–457
(see also Zyre project)
GSL, 387–392

H

Hadoop Zookeeper project, 24
Hangman role, 369
hardware
failure of, 142
writing messages to hard disk, 195–206

Harmony pattern, 452–454
heartbeating, 116, 125, 159–163
in Zyre project, 454–455
not using, 160–160
one-way heartbeats, 160
for Paranoid Pirate pattern, 161–163
in Paranoid Pirate pattern, 151, 151, 154, 155, 156, 157, 158, 158
ping-pong heartbeats, 161–163
Hello World example, 5–9
high-level API for ZeroMQ, 102–110
high-level patterns, 38
high-water mark (HWM), 77–78
Historian role, 369
HTTP protocol, using ZeroMQ for, 35–36
HWM (see high-water mark)

I

I/O threads, 36
IANA (Internet Assigned Numbers Authority), 424
idempotent services, 193–194
identity, 89–91
iMatix, xiii, 327
innovation, models for, 356–364
inproc (inter-thread) transport, 35, 64, 68–70
binding order requirement for, 35
high-water mark with, 78
Inter-Broker Routing example, 116–140
brokers, interconnecting, 119–122
cloud flow for, 126–133
clusters of workers and clients for, 117–121
final code for, 133–140
ipc transport for, 123
limitations of, 140
local flow for, 126–133
requirements for, 116–117
sockets, naming, 122–123
state flow for, 123–126
inter-process transport (see ipc transport)
inter-thread transport (see inproc transport)
intermediation, 45
for publish-subscribe pattern, 46–47
for request-reply pattern, 48–54
zmq_proxy() function for, 53–54
Internet Assigned Numbers Authority (IANA), 424
Internet of Things, 425
interrupt signals, handling, 61–62
ipc (inter-process) transport, 35
binding to same endpoint twice, 33
for Inter-Broker Routing, 123

J

Jakl, Michael (contributor), 480
JeroMQ implementation, 326
JSON, 383

L

last value caching (LVC), 250–254
late (slow) joiners
with Clone pattern, 264, 271
with FileMQ project, 421–422
with Harmony pattern, 460
with pipeline pattern, 20
with publish-subscribe pattern, 14
TCP for, 448
Laughing Clown role, 367
Laxy Pirate pattern, 144–148
Lazy Perfectionist role, 366
van Leeuwen, Tom (contributor), 479
LGPL license
for examples in this book, 484
for ZeroMQ, 326
Libero, 403–410
libzmq library, 326
bindings for, 326
reimplementations of, 326
upgrading to version 3.2, 27
licensing
BSD license, 330–332, 335
for examples in this book, 484
GPL license, 330–332, 335, 339, 378
LGPL license, 326, 484
MIT/X11 license, 484
Load-Balancing Message Broker example, 96–102
load-balancing pattern, 91–102
CZMQ for, 105–110
DEALER and ROUTER combination for, 94–102
REQ and ROUTER combination for, 92–94, 117–121
logging, in distributed environment, 467–473
LVC (last value caching), 250–254

M

Majordomo Management Interface (MMI), 192
Majordomo pattern, 38, 164–186
Majordomo pattern, Asynchronous, 186–191
MDP (Majordomo Protocol), 164–165, 185, 382
memory leaks, detecting, 62–63
message queues, overflowing, 37, 142, 247, 255
Message-Oriented Pattern for Elastic Design (MOPED), 372–375
MessagePack, 384–385
messages, 39–41
  benefits of ZeroMQ for, 23–27
  best practices for, 22
  content of, accessing, 39
  envelopes for, 75–76, 81–86
  flow control for, 77
  high-water mark for, 77–78
  losing, causes of, 78
  multipart, 40–41, 44–45, 382
  patterns for sending (see patterns)
  reading, 39
  receiving, 32, 34–35
  releasing, 39
  sending, 32, 34–35, 39, 40
  size of, 39
  string format for, 10–11
  as structures, 32, 39
  TCP for, 23
  writing, 39
  zero-length, 41
Mindful General role, 367
MIT/X11 license, 484
MMI (Majordomo Management Interface), 192
monitoring, in distributed environment, 467–473
MOPED (Message-Oriented Pattern for Elastic Design), 372–375
multicast messaging (see publish-subscribe pattern)
multicast transports, 35
multipart messages, 40–41, 44–45, 382
  envelopes for, 75–76
  high water mark for, 78
  zero-copy used with, 74
Multiple Socket Reader/Poller example, 41–44
Multithreaded Hello World example, 65–67
Multithreaded Relay example, 68–70
multithreading, 63–67
  best practices for, 64
  for client stack, 310–320
  exiting, best practices for, 22
  I/O threads, 36
  for increasing subscriber speed, 258–260
  portable thread management, 104
  signaling between threads, 68–70
Mystic role, 369

N

Nagle's algorithm, 186
networks
  asynchronous disconnected, 194–206
  failure of, 142
  plugging sockets into, 32–34
nodes, coordination between, 70–74
  (see also client node; server node)
non-blocking reads, 41
non-blocking request-reply, 48–54

O

one-way data distribution pattern (see publish-subscribe pattern)
one-way heartbeats, 160
Open Door role, 367
open source software
  licensing for, 330–332
  models for, 325, 328
  reasons for, 325, 328
OpenAMQ server, 206
optimization
  hand-optimizing high-volume data flows, 385
  with heartbeats, 160
  zero-copy for, 74–74

P

PAIR socket, 38, 68–70
Parallel Task Ventilator example, 16–20, 57–59
Paranoid Pirate pattern, 151–159, 161–163
path hierarchy, 282
patterns, 37–38
  asynchronous client/server (see asynchronous client/server pattern)
  Asynchronous Majordomo pattern, 186–191
  Binary Star pattern, 206–222
  Black Box pattern, 258–260
  Clone (see Clone pattern)

Espresso pattern, 247–249
exclusive pair (see exclusive pair pattern)
Freelance pattern, 223–243
group messaging pattern, 455–457
Harmony pattern, 452–454
high-level, 38
interoperability between, protocols for, 163
Lazy Pirate pattern, 144–148
load-balancing (see load-balancing pattern)
Majordomo pattern, 164–186
Paranoid Pirate pattern, 151–159, 161–163
pipeline (see pipeline pattern)
publish-subscribe (see publish-subscribe pattern)
request-reply (see request-reply pattern)
Simple Pirate pattern, 148–151
Suicidal Snail pattern, 254–257
Titanic pattern, 194–206
peer connectivity
blocked peers, handling, 464–466
Harmony pattern for, 452–454
peering interconnection, 121
pgm transport, 246, 250
ping-pong heartbeats, 161–163
pipeline pattern, 37
examples of, 16–20
fair-queuing with, 20
for file transfers, 400–403
reliability for, 143
shutting down cleanly, 57–59
slow joiner syndrome with, 20
Pirate Gang role, 368
Pirate Pattern Protocol (PPP), 163, 164
pirate patterns, 143
Asynchronous Majordomo pattern, 186–191
Binary Star pattern, 206–222
Freelance pattern, 223–243
interoperability between, protocols for, 163
Lazy Pirate pattern, 144–148
Majordomo pattern, 164–186
Paranoid Pirate pattern, 151–159, 161–163
Simple Pirate pattern, 148–151
Titanic pattern, 194–206
port numbers, registered, 424
PPP (Pirate Pattern Protocol), 163, 164
preemptive discovery, 432
production of this book, xiii–xv, 481–484
programming practices, 21–23
projects
FileMQ (see FileMQ project)
using ZeroMQ, list of, 327
Zyre (see Zyre project)
protocol assertions, 470–471
protocol specifications (see unprotocols)
Provocateur role, 369
proxy or broker, 45, 89
(see also server node)
multiple, interconnecting, 119–122
for publish-subscribe pattern, 46–47
for request-reply pattern, 48–54
zmq_proxy() function for, 53–54
psychology of software architecture, 328–330, 333–335, 366–369
PUB (publish) socket, 14–15, 38
publish-subscribe pattern, 37, 245–247
bridging transports, 54–56
clients sharing state for (see Clone pattern)
envelopes used with, 75–76
examples of, 11–15
extended to use a proxy, 46–47
filtering with, 15
group messaging pattern as, 455–457
increasing subscriber speed for, 258–260
last value caching (LVC) for, 250–254
one-way heartbeats for, 160
reliability for, 143, 247
(see also Black Box pattern; Clone pattern; Espresso pattern; Suicidal Snail pattern)
scalability of, 246
for sending kill messages, 57–59
slow joiner syndrome with, 14
slow subscribers, handling, 254–257
synchronizing publisher and subscriber, 70–74
synchronizing publisher and subscriber in, 15
tracing network for, 247–249
publisher (see server node)
PULL socket, 18–20, 38
(see also pipeline pattern)
PUSH socket, 16–20, 38
(see also pipeline pattern)
PyZMQ binding, 326

Q

queue proxy, 148–151, 152

R

reactor
Binary Star pattern as, 218–222
Clone pattern using, 292–295
replacing zmq_poll() function with, 104, 108
registered port numbers, 424
reliability, 141–142
Freelance Pattern for, 223–243
for pipeline pattern, 143
for publish-subscribe pattern, 143, 247
(see also Black Box pattern; Clone pattern; Espresso pattern; Suicidal Snail pattern)
for request-reply pattern, 142–144
(see also pirate patterns)
reliable request-reply (RRR) patterns (see pirate patterns)
REP (reply) socket, 5–9, 38, 85, 86–88
REQ (request) socket, 8–9, 38, 85, 86–88
REQ and ROUTER combination, 87
load balancing using, 92–94
load-balancing using, 117–121
Request-Reply Broker example, 50–54
request-reply pattern, 37
asynchronous client/server pattern as, 111–116
example of, 5–9
extended to use proxy, 48–54, 82–85
load-balancing pattern as, 91–102, 117–121
reliability for, 141–144
(see also pirate patterns)
reply messge envelopes with, 81–86
synchronizing subscribers and publisher, 71–74
valid socket combinations for, 86–88
RFC 2234, 379
Rolling Stone role, 368
ROUTER and ROUTER combination, 88
ROUTER socket, 38, 49, 85, 86, 86–88, 89–91
ROUTER-DEALER proxy, 48–54, 82–85
(see also DEALER and ROUTER combination)
RRR (reliable request-reply) patterns (see pirate patterns)

S

SASL (Simple Authentication and Security Layer), 410–411
scalability
of publish-subscribe pattern, 246
of sockets, 27–27
serialization of data, 382–392
code generation for, 386–392
framing for, 382
handwritten binary serialization, 385–386
languages for, 383–384
libraries for, 384–385
server node
binding sockets to endpoint, 32–34
high-availability pair of, 206–222
multiple, clients connecting to with proxy, 143
(see also Majordomo pattern; Paranoid Pirate pattern; Simple Pirate pattern; Titanic pattern)
multiple, clients connecting to without proxies, 144, 223–243
role of, 32, 33, 89
single, clients connecting to, 143, 144–148
services
discovering, 191–193
idempotent, 193–194
sessions, for ephemeral values, 284
Shalt, Vadim (contributor), 480
shared queue, 48–54
SIGINT (Ctrl-C), handling, 61–62, 105, 107, 110, 110
signals, handling, 61–62
SIGTERM, handling, 61–62
Simple Authentication and Security Layer (SASL), 410–411
Simple Pirate pattern, 148–151
slow joiners (see late joiners)
Social Architecture (Hintjens), 328
Social Engineer role, 367
sockets, 32–37
(see also specific socket types)
best practices for, 22
closing automatically, 104
combinations of, 38, 86–89
configuring, 32
connections between, creating, 32–34
creating, 32
destroying, 32
high-water mark handling by, 77
identity of, 89–91
life cycle of, 32

messages carried by (see messages)
multiple connections managed by, 28
multiple, reading from, 41–44
non-blocking reads of, 41
as not threadsafe, 64
scalability of, 27–27
types of, 34–34, 38
as void pointers, 32
software architecture
C4 contract for, 325, 335
connectedness, 3–4
design models for, 356–364
guidelines for, 327–335
MOPED pattern for, 372–375
open source model for, 325, 328
psychology of, 328–330, 333–335, 366–369
serialization of data, 382–392
SO_REUSEADDR option, 439
specifications (see unprotocols)
split-brain syndrome, 211
state
flow of, for Inter-Broker Routing, 123–126
maintaining, in FileMQ project, 417–418
sharing (see Cone pattern)
state machines, 403–410
string format, 10–11
SUB (subscribe) socket, 14–15, 38
(see also publish-subscribe pattern)
subscriber (see client node)
subtrees, 281
Suicidal Snail pattern, 254–257
Synchronized Publisher example, 71–74
s_recv() function, 10
s_send() function, 11

T

TCP
connections, compared to ZeroMQ, 33
for late joiners, 448
messaging using, 23
sockets, compared to ZeroMQ, 34
tcp transport, 35
testing
assertions for, 457
best practices for, 21
up-front testing, 458–459
for Zyre project, 457, 459–463
threads (see multithreading)
time to live (TTL), for ephemeral values, 284
Titanic pattern, 194–206
TOD (Trash-Oriented Design), 359–361
topic tree, 282
transports, 33, 35, 35
(see also specific transports)
bridging, 54–56
multicast, 35
unicast, 35
Trash-Oriented Design (TOD), 359–361
TTL (time to live), for ephemeral values, 284

U

UDP
beacon message format with, 450–452
cooperative discovery using, 434–438
discovery using, 448–448
udplib library, 438
unicast transports, 35
unprotocols, 163, 375–382
for FileMQ project, 413–414
for Zyre project, 475–475
up-front testing, 458–459

V

valgrind, 62–63

W

Weather Update Proxy example, 54–56
Weather Update Server example, 11–15
website resources
ABNF, in RFC 2234, 379
C4 contract, 325, 335
CHP (Clustered Hashmap Protocol), 306
code examples, xv
Digital Standards Organization (Digistan), 328
FileMQ project, 414
FILEMQ protocol, 414
for this book, xvi
Fork + Pull Model, GitHub, 335
Foundation for a Free Information Infrastructure (FFII), 328
IANA, 424
iMatix, xiii
MessagePack, 384
Nagle's algorithm, 186
OpenAMQ server, 206

PPP (Pirate Pattern Protocol), 163, 164
public ZeroMQ contracts, 163
PyZMQ binding, 326
SASL, 410
Social Architecture (Hintjens), 328
udplib library, 438
ZeroMQ community, 327
ZRE protocol, 475
WiFi, 427–432
worker (see client node)

X

XML, 383–384
XPUB (extended publisher) socket, 46, 54–56
XPUB-XSUB proxy, 46–47
XSUB (extended subscriber) socket, 46, 54–56

Z

zero-copy, 74–74
zero-length messages, 41
ZeroMQ, xiii, 3–4, 23–27
best practices for, 21–23
community for (see community)
high-level API for, 102–110
version of
assumed for this book, 5
determining, 11
upgrading to version 3.2, 27
zhelpers.h file, 11, 39
zloop reactor, CZMQ, 108–110
zmq_bind() function, 32–34
zmq_connect() function, 32–34
zmq_ctx_destroy() function, 21, 28
hanging, 22
non-fatal errors from, 57
zmq_ctx_new() function, 21, 28
zmq_ctx_set() function, 28, 36
zmq_ctx_set_monitor() function, 28
ZMQ_DONTWAIT option, 27, 57
zmq_errno() function, 56
ZMQ_IDENTITY option, 89
zmq_init() function, 28
ZMQ_IO_THREADS option, 36
zmq_msg_close() function, 39
zmq_msg_copy() function, 40
zmq_msg_data() function, 39
zmq_msg_init() function, 39, 40
zmq_msg_init_data() function, 41, 74–74
zmq_msg_init_size() function, 39
zmq_msg_recv() function, 6, 28, 34–35, 39, 57, 62
zmq_msg_send() function, 6, 28, 34–35, 39, 40, 62
zmq_msg_size() function, 39
zmq_msg_t objects, 39
ZMQ_NOBLOCK option, 27
zmq_poll() function, 41–44
interrupts affecting, 62
with multipart messages, 45
replacing with a reactor, 104, 108
zmq_proxy() function, 53–54, 247–249
ZMQ_RCVMORE option, 45
zmq_recv() function, 27
ZMQ_ROUTER_MANDATORY option, 91
ZMQ_ROUTER_RAW option, 36
zmq_send() function, 27
zmq_setsockopt() function, 14, 89
zmq_strerror() function, 56
ZMQ_SUBSCRIBE option, 255
zmq_term() function, 28
zmq_version() function, 11
ZMTP protocol, 40
ZRE protocol, 449, 475–475
Zyre project, 448–450
assertions for, 457, 460
beacon message format for, 450–452
blocked peers, handling, 464–466
content distribution, 473–475
disconnections, heartbeating detecting, 454–455
FileMQ project used in, 449
future directions for, 476
group messaging in, 455–457
logging and monitoring for, 467–473
peer connectivity, Harmony pattern for, 452–454
testing, 457–466
ZRE protocol for, 449, 475–475

ZeroMQ：云时代极速消息通信库

请潜心研究ØMQ（又名ZeroMQ）这个智能套接字库，它让你的应用程序能够获得快速、简便、基于消息的并发性。有了这本快节奏的指南，你将在实践中学习如何使用这个可扩展、轻量级且高度灵活的网络工具，从而在集群、云服务端等各种多系统环境之间交换消息。

ØMQ的维护者Pieter Hintjens带你观察现实世界的应用程序，并用C语言编写的扩展例子帮助你使用ØMQ的API、套接字和模式。了解如何使用特定的ØMQ编程技术，构建多线程应用程序，并创建自己的消息传递架构。你会学到ØMQ如何与多种编程语言和大多数操作系统共用，只有很少的成本或根本没有成本。

- 了解ØMQ的主要模式：请求-应答、发布-订阅和管道
- 通过建立几个小应用程序来使用ØMQ套接字和模式
- 通过工作实例探索ØMQ的请求-应答模式的高级使用
- 构建一个在代码或硬件出现故障时保持工作可靠性的请求-应答模式
- 扩展ØMQ的核心发布-订阅模式的性能、可靠性、状态分发与监控
- 了解用ØMQ来构建分布式架构的技术
- 探索为分布式应用程序建立一个通用的框架有什么要求

"本年度你能读到的最好的爱与套接字的故事。"

—— Alexis Richardson
VMware的高级主管和
新人训练师

Pieter Hintjens，iMatix公司的CEO和首席软件设计师，该公司创建了ØMQ。他是自由信息基础设施基金会（FFII）的前任会长，欧洲专利大会和数字标准组织的创办人，Wikidot公司的前CEO，他还是ØMQ的一位维护者。

图书分类：C/C++

策划编辑：张春雨
责任编辑：刘舫

O'REILLY®
oreilly.com.cn



Broadview®
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)



ISBN 978-7-121-25311-9

定价：108.00元